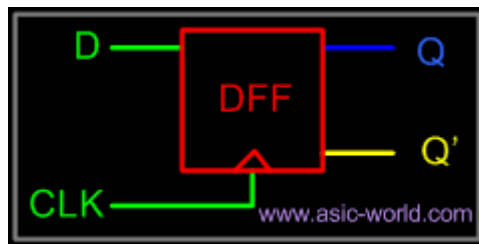


## INTRODUCTION

Verilog is a **HARDWARE DESCRIPTION LANGUAGE (HDL)**. A hardware description language is a language used to describe a digital system: for example, a network switch, a microprocessor or a memory or a simple flip-flop. This just means that, by using a HDL, one can describe any (digital) hardware at any level.



```

1 // D flip-flop Code
2 module d_ff ( d, clk, q, q_bar);
3 input d ,clk;
4 output q, q_bar;
5 wire d ,clk;
6 reg q, q_bar;
7
8 always @ (posedge clk)
9 begin
10   q <= d;
11   q_bar <= !d;
12 end
13
14 endmodule
  
```

You could download file d\_ff.v [here](#)

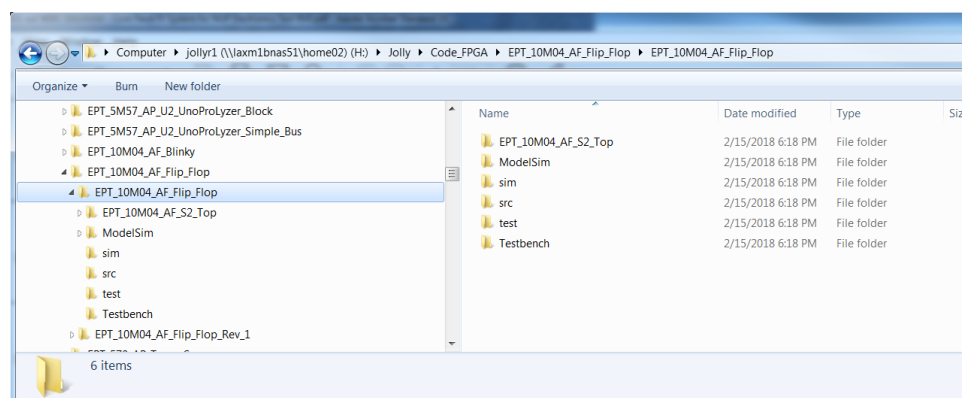
One can describe a simple Flip flop as that in the above figure, as well as a complicated design having 1 million gates. Verilog is one of the HDL languages available in the industry for hardware designing. It allows us to design a Digital design at Behavior Level, Register Transfer Level (RTL), Gate level and at switch level. Verilog allows hardware designers to express their designs with behavioral constructs, deferring the details of implementation to a later stage in the final design.

### ////////////////////////////////Lesson #1////////////////////////////////

In this first lesson, let's set up the ModelSim environment and simulate a simple flip flop.

First, set up your file system to allow easy access to updating and modifying your files. There are many ways to set up your file system, the following is the one that works best for me.

Copy the Lesson 1 Folder over to our local drive from the DVD. In this folder you will notice a top level project folder.



The project level folder is called “EPT\_10M04\_AF\_Flip\_Flop”. Under this project level folder are various revisions of the project. Multiple revisions allow you to keep copies of previously working projects and refer to them later when the currently edited source code no longer compiles.

Under the current revision of the project, this folder does not include a “\_Rev\_x” suffix. The project is organized with the following folders:

- EPT\_10M04\_AF\_S2\_Top – This folder contains all the Altera project level files such as pin, sdc, configuration, programming object files etc.
- ModelSim – This folder contains all the compiled object files for ModelSims use. It also includes the \*.do files. These ‘do’ files are the make files for ModelSim. They tell the compiler which source files to compile and allow compile time options.

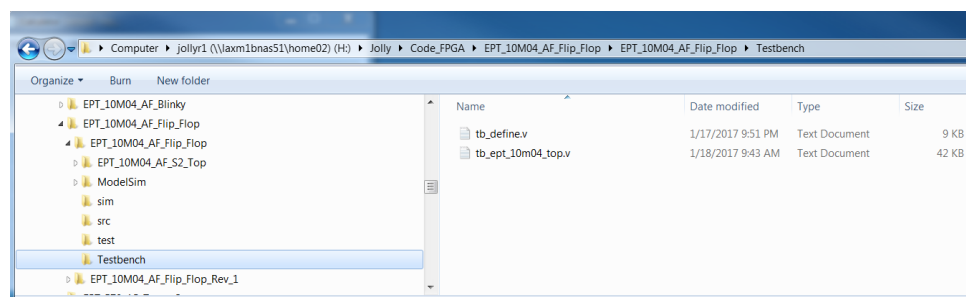
- Sim – This folder contains special source files that have non-synthesizeable constructs in them.
- Src – This folder contains all your source files. Only synthesizeable code should go into this folder.
- Test – This folder contains source code for the models which are used to model behavior of devices that are not part of the FPGA. These source files can contain non-synthesizable code.
- Testbench -- This folder contains the main testbench source code. The testbench controls the operation of the user source code, test models and simulation code. It provides the main stimulus such as the clock and reset.

In the src folder, create a file named “EPT\_10M04\_AF\_S2\_Top.v”. Then open this file in an editor, I prefer to use NotePad++. Add in the D flip-flop code and add comments to describe the file and the parts of the file.

```

EPT_10M04_AF_S2_Top.v x
20  ///< Revision History:
21  ///<      DATE      VERSION  DETAILS
22  ///<      1/15/17    1        Created
23  ///<
24  ///<
25  ///<
26  ///<#####
27
28
29  ///<#####
30  ///< Module Declaration
31  ///<#####
32
33  ///< D flip-flop Code
34  2 module EPT_10M04_AF_S2_Top ( d, clk, q, q_bar);
35  3 input d ,clk;
36  4 output q, q_bar;
37  5 wire d ,clk;
38  6 reg q, q_bar;
39  7
40  8 always @ (posedge clk)
41  9 begin
42  10   q <= d;
43  11   q_bar <= ! d;
44  12 end
45  13
46  14 endmodule
47
  
```

Next, setup the Testbench file to provide a stimulus for our user code. The Testbench file provides all the hardware external devices that the FPGA needs to operate the user code. These include the oscillator/clock, reset, push buttons, communications, etc... Open the Testbench folder and notice the two files in it.



- Tb\_define.v contains pre-defined parameters for the testbench execution
- Tb\_ept\_10m04\_top.v contains declarations, stimulus, tasks and leif modules to exercise the user code.

```

tb_ept_10m04_top.v
13  //#      DATE      VERSION  DETAILS
14  //#      01/17/2017  A      Created
15  //#
16  //
17  //
18  //#####
19
20  `timescale 1ns/1ps
21
22  `include "../Testbench/tb_define.v"
23
24  module tb_ept_10m04_top;
25
26  //-----
27  // Parameter Declarations
28  //-----
29  parameter BEGIN_SECTION      = 8'h00;
30  parameter TRIGGER_IN_SECTION = 8'h01;
31  parameter TRANSFER_IN_SECTION = 8'h02;
32  parameter BLOCK_OUT_SECTION  = 8'h03;
33  parameter WRITE_SECTION      = 8'h04;
34  parameter BLOCK_IN_SECTION   = 8'h05;
35  parameter ARDUINO_WRITE_TO_FIFO = 8'h06;
36  parameter ARDUINO_ANALOG_MONITOR = 8'h07;
37  parameter LED_BLINKY         = 8'h08;
38  parameter READ_SECTION_MODEL = 8'h09;
39  parameter WRITE_ADC_CONFIG_REG = 8'h0a;
40  parameter SEND_ADC_CONV_START = 8'h0b;
41  parameter DSO BEGIN_SECTION  = 8'h0c;
  
```

Inside the tb\_ept\_10m04\_top.v file, we see 'module' declaration along with the name of test bench. The parameter declarations are listed to allow certain registers to have constant values. We will discuss parameters and the details of the Verilog files later in the tutorial. For now, we will focus on getting started and performing our first simulation. Scroll down the testbench file and locate the "Instantiate DUT" section.

```
tb_ept_10m04_top.v
1273 task stopsim;
1274     begin
1275         $fdisplay(logfile,"Simulation Finished at time %0t\n",$time);
1276         $stop;
1277         $fclose(logfile);
1278     end
1279 endtask // stopsim
1280
1281
1282 //-----
1283 // Instantiate DUT
1284 //-----
1285
1286 EPT_10M04_AF_S2_Top      DUT
1287 (
1288     .d            (d) ,
1289     .clk          (clk_50) ,
1290     .q            (q) ,
1291
1292     .q_bar        (q_bar)
1293
1294 );
1295
1296
1297 endmodule // tb_ept_10m04_top.v
1298
1299
```

You can see that the testbench uses the name of the module declared in 'EPT\_10M04\_AF\_S2\_TOP.v'. This is called the leaf instantiation. When ModelSim starts the compilation process, it will search the directory path for the module 'EPT\_10M04\_AF\_S2\_TOP'. This instantiation must include the inputs and outputs declared in the module. In this case:

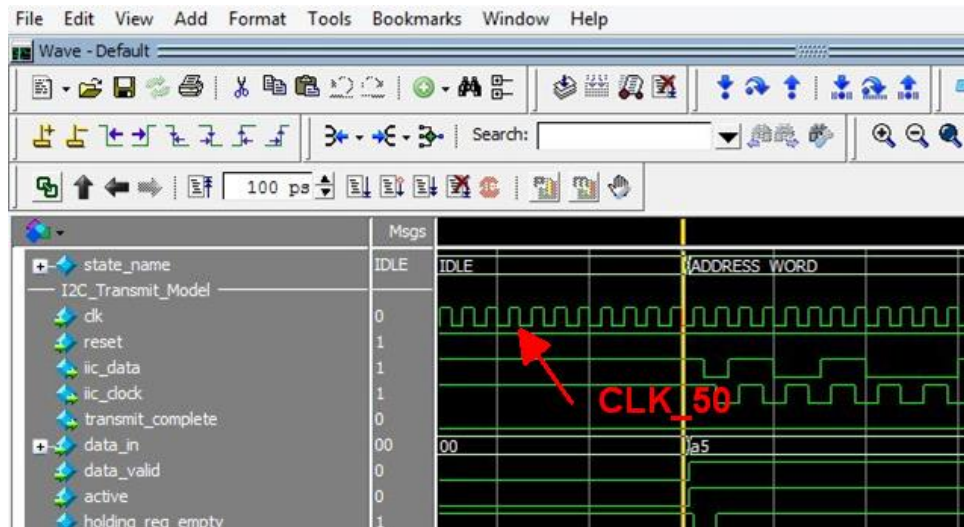
- 'd' – Input into the D flip flop
- 'clk' – Input into the D flip flop
- 'q' – Output from the D flip flop
- 'q\_bar' – Output from the D flip flop

Using the leaf instantiation module, we have now connected the testbench with the user code. When ModelSim starts the simulation process, testbench will control the stimulus to the inputs and accept the outputs from the user code.

To add stimulus, we use Verilog code. To add a clock, we use the 'forever' key word.

```
tb_ept_10m04_top.v
286     clk_66 = 0;
287     forever
288     //begin
289     # `CYCLE_66 clk_66 = !clk_66;
290     //end
291     end
292 //-----
293 //  Generate Internal Clock with 50 MHz
294 //-----
295
296     initial
297     begin
298         clk_50 = 0;
299         forever
300         //begin
301         # `CYCLE_50 clk_50 = !clk_50;
302         //end
303     end
304
```

The parameter 'CYCLE\_50' is defined in the tb\_define.v file. We add the '#' character at the beginning of the line to inform the compiler to "add the following" as a delay in simulator steps. During simulation, the simulator will start a timer that halts this signal (and only this signal) and waits for the timer to expire. After the delay has expired, the signal is set equal to its complement state. Then, because of the 'forever' keyword, the process starts over, delay, then set signal to its complement. The result is a clock at 50MHz. We will go through the details of all these details later in the tutorial.



Next, add the stimulus for 'd' input. We do this using the 'initial' block. Everything between the 'begin' 'end' keywords is executed once per simulation.

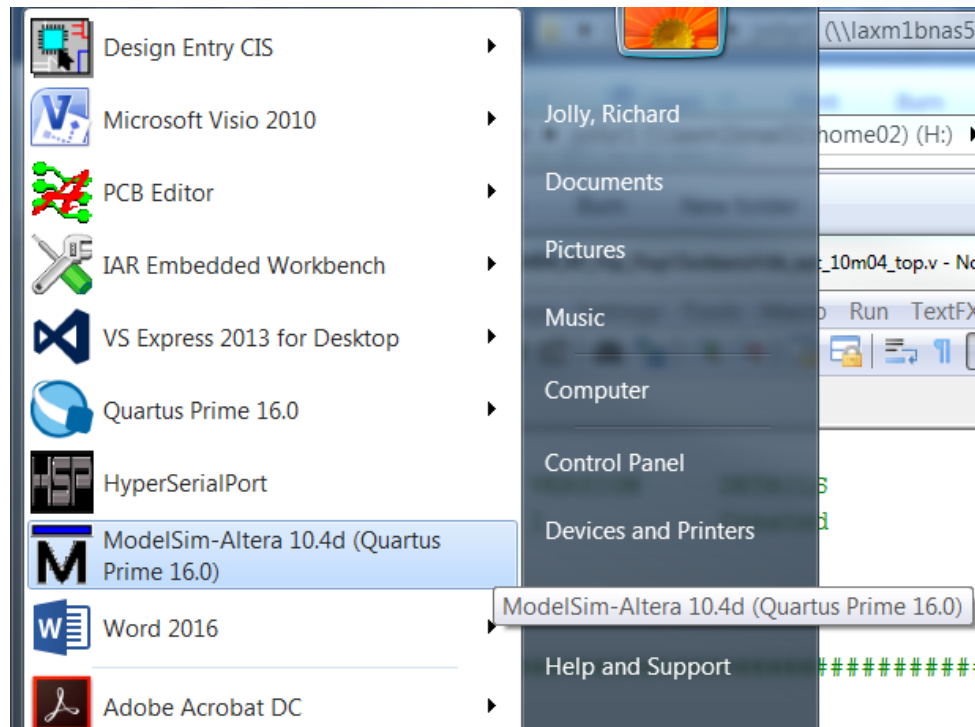


```
tb_ept_10m04_top.v
315
316 //-----
317 //  Apply Stimulus
318 //-----
319
320 initial
321 begin
322     //////////////////////////////////////
323     //Print the Title of the Section that is being tested
324
325     call_title(D_FLIP_FLOP);
326     #(100 * `CYCLE)
327     d = 1'b1;
328
329     #(100 * `CYCLE)
330     d = 1'b0;
331
332     #(50 * `CYCLE)
333     d = 1'b1;
334
335
336     // End of Simulation
337     #(5000000 * `CYCLE);
338     stopsim;
339
340
341 end //stimulus
```

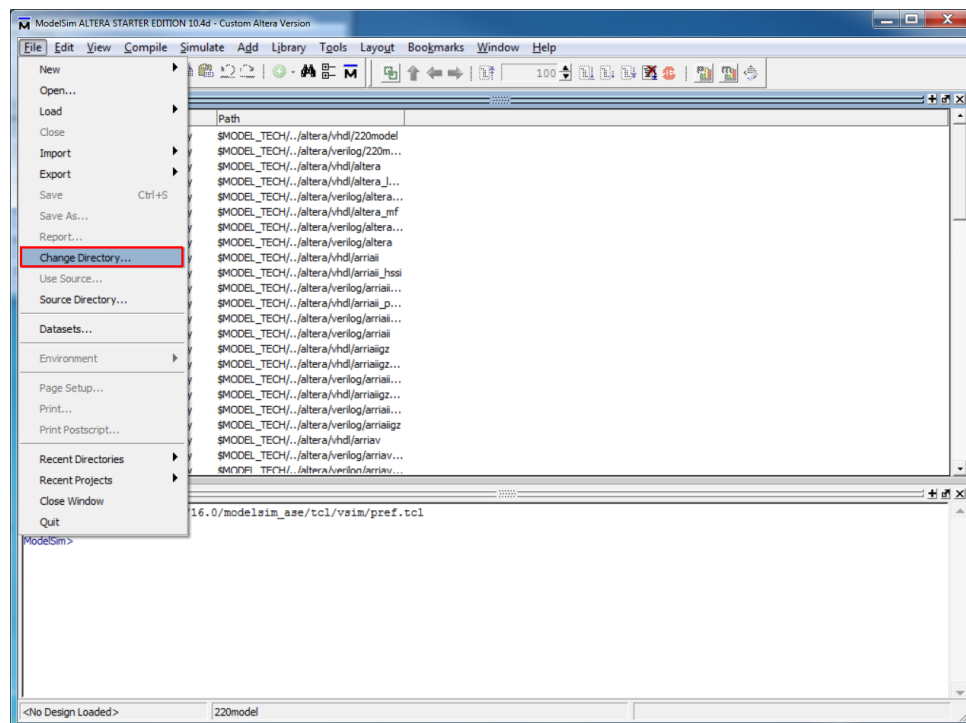
You can see here that the stimulus 'd' is set to 1'b1 after a delay of 100 \* CYCLE. Where 'CYCLE' is defined in the tb\_define.v file. Then, after another delay, the 'd' is set to 1'b0. Finally, a delay of 50\*CYCLE is added then the 'd' is set to 1'b1. The result is a toggle from high to low to high on the 'd' signal.

Next, get the ModelSim loaded up on your laptop/PC. Follow the install guide on how to install the ModelSim application.

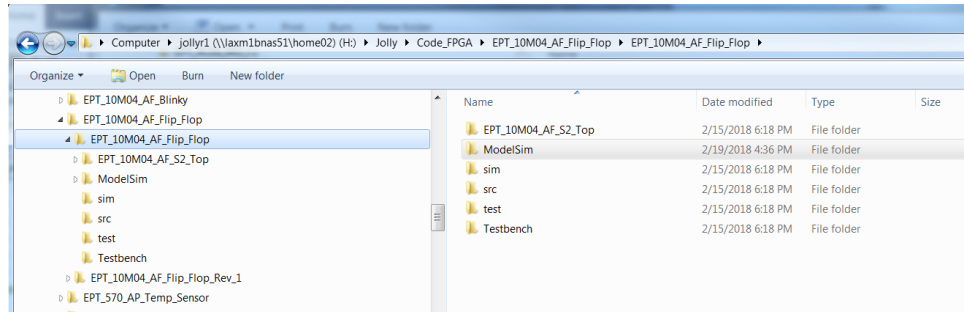
## Verilog Programming Guide



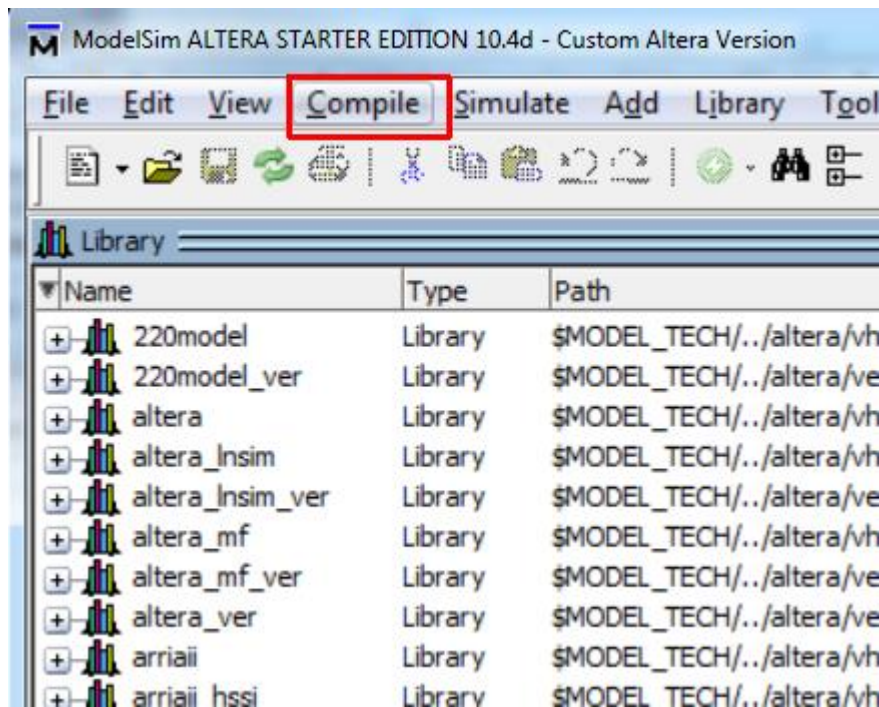
Then, open the application and Go to File->Change Directory



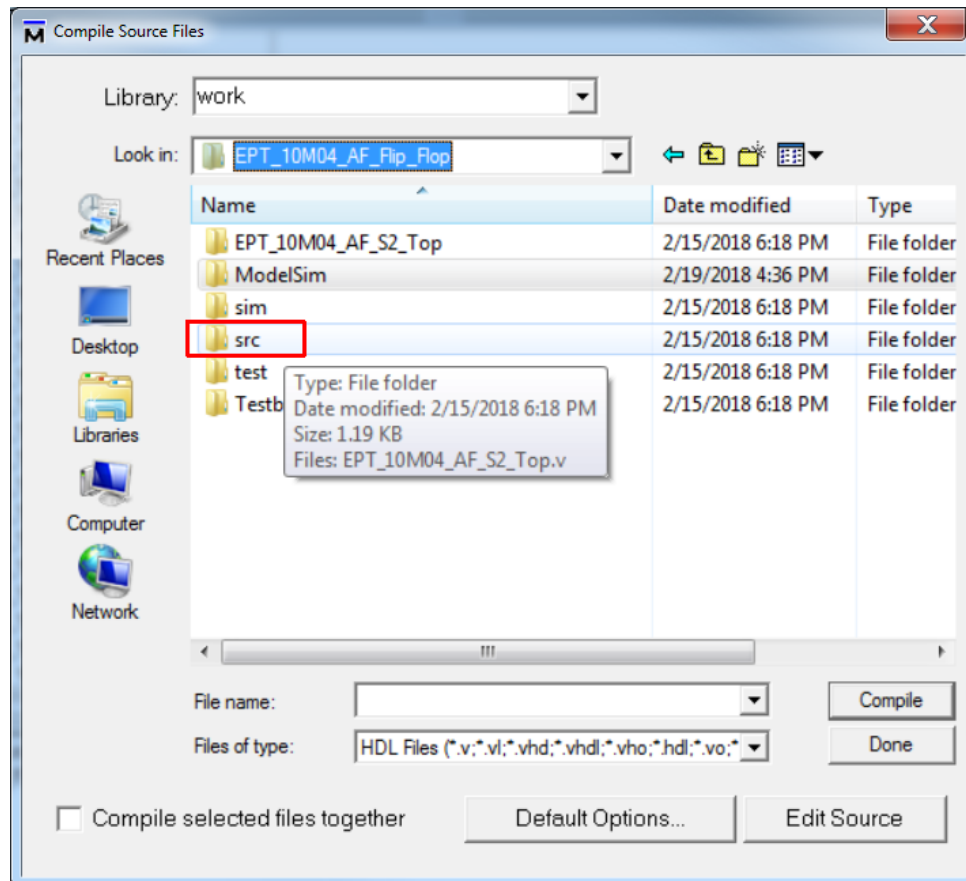
At the dialog box, locate the ModelSim folder under the EPT\_10M04\_AF\_S2\_TOP project.



Select the ModelSim folder. Next, we will compile each module individually. Click on the 'Compile' menu item. Select the 'Compile' tab.



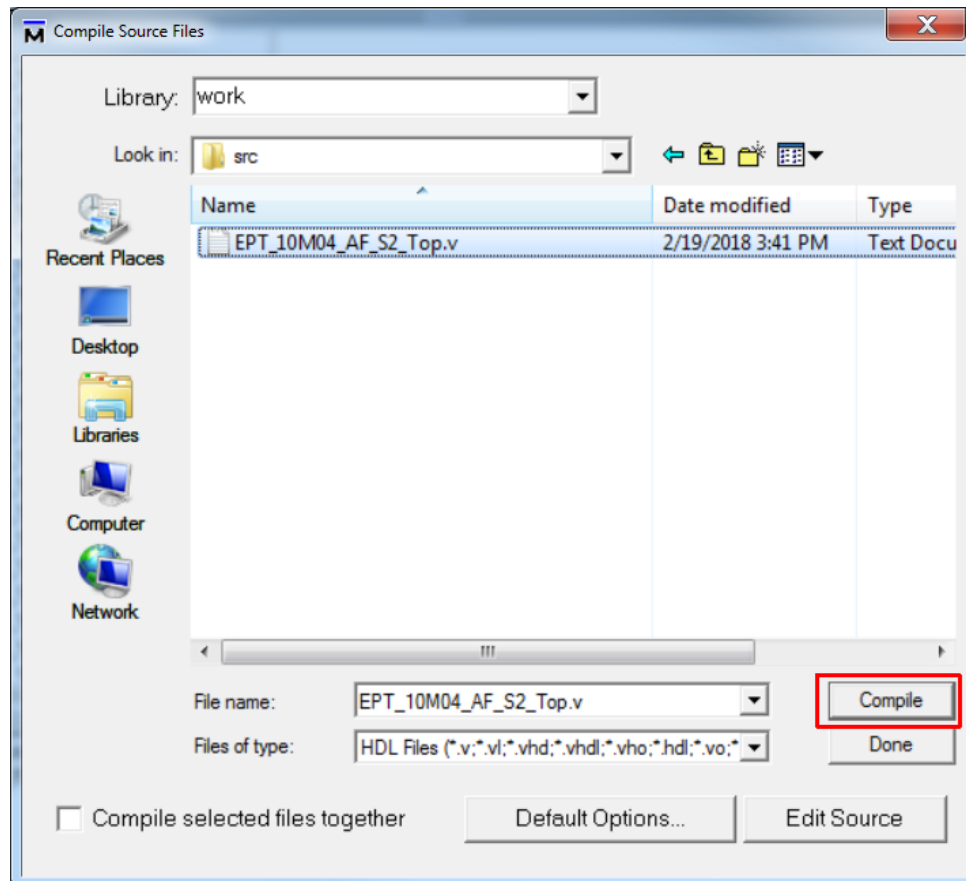
In the 'Compile Source Files' window, select the 'src' folder.



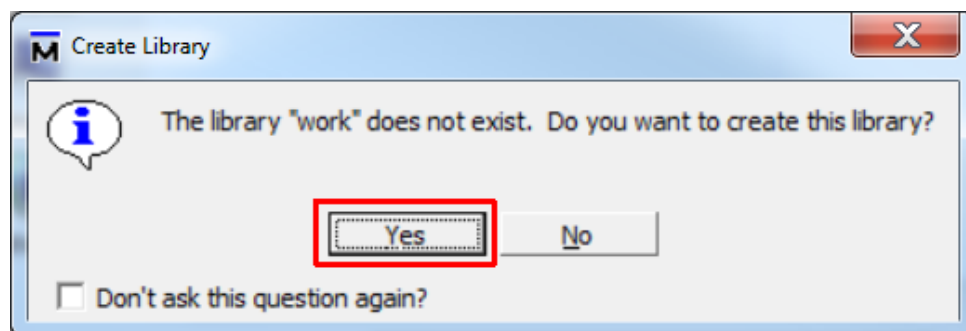
Select the 'EPT\_10M04)AF\_S2\_Top.v' file.



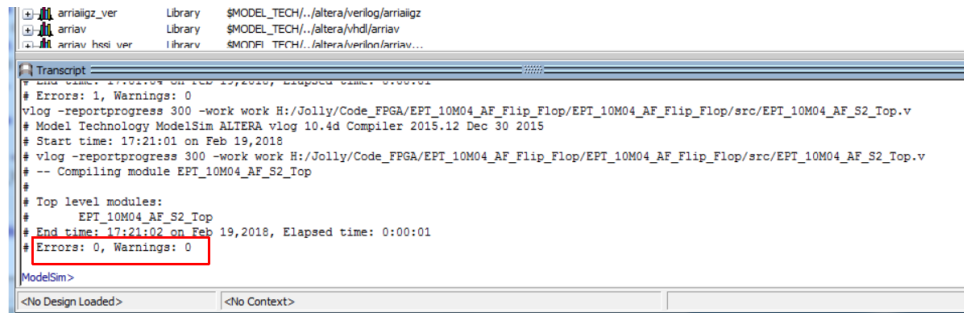
Click the 'Compile' button.



You will receive the 'Create Library' message box. Select Yes.



After the file is compiled, the log will indicate the status of compilation.



```

Library $MODEL_TECH/./altera/verilog/arria10z
Library $MODEL_TECH/./altera/vhdl/arriav
Library $MODEL_TECH/./altera/verilog/arriav...

Transcript
# Errors: 1, Warnings: 0
vlog -reportprogress 300 -work work H:/Jolly/Code_FPGA/EPT_10M04_AF_Flip_Flop/EPT_10M04_AF_Flip_Flop/src/EPT_10M04_AF_S2_Top.v
# Model Technology ModelSim ALTERA vlog 10.4d Compiler 2015.12 Dec 30 2015
# Start time: 17:21:01 on Feb 19, 2018
vlog -reportprogress 300 -work work H:/Jolly/Code_FPGA/EPT_10M04_AF_Flip_Flop/EPT_10M04_AF_Flip_Flop/src/EPT_10M04_AF_S2_Top.v
# -- Compiling module EPT_10M04_AF_S2_Top
#
# Top level modules:
#   EPT_10M04_AF_S2_Top
# End time: 17:21:02 on Feb 19, 2018, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
ModelSim>
<No Design Loaded> <No Context>

```

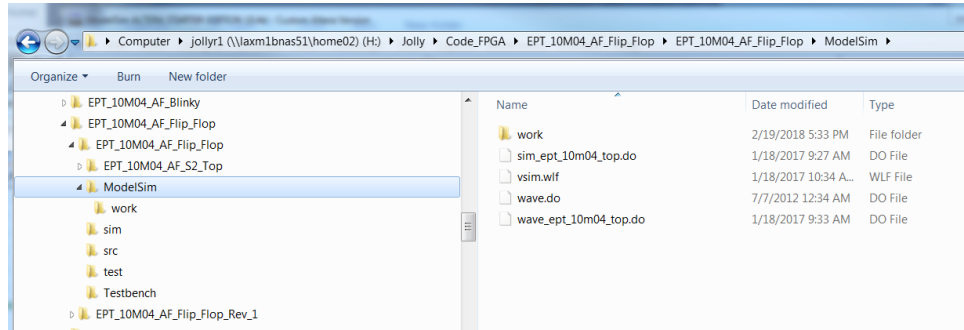
Next, repeat the compile steps for the

- tb\_define.v
- tb\_ept\_10m04\_top.v



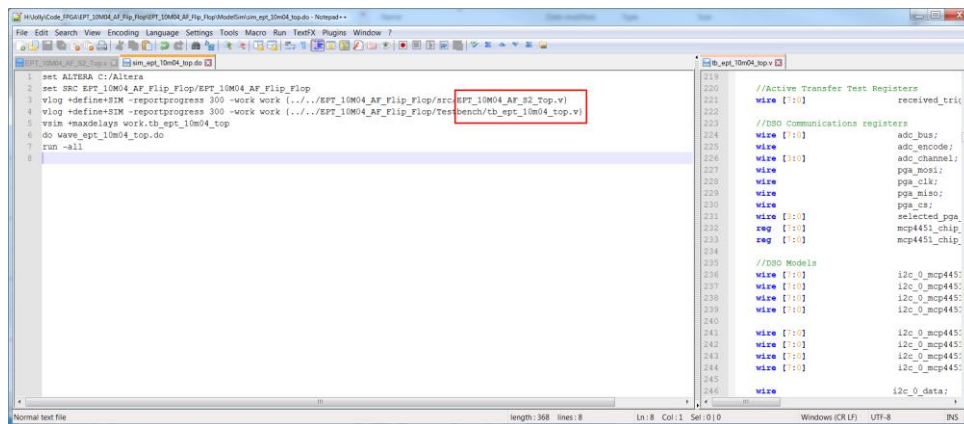
Note, you will not get receive the message box asking to create the

'work' folder this time. After the compilation has completed, we are ready to run the Simulation. The simulation environment is controlled by the \*.do files. The \*.do files act as makefiles to control which files get simulated and add compile options. There are two \*.do files needed for our simulation. They are found in the ModelSim folder.



- Sim\_ept10m04\_top.do – Contains the files and compile options.
- Wave\_ept10m04\_top.do – Contains the display flags for the 'Waveforms' window.

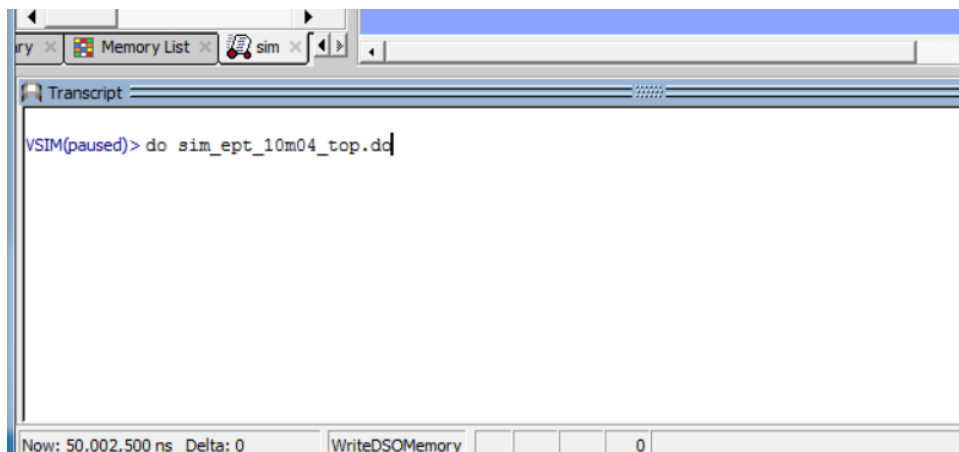
When we look inside the sim\_ept\_10m04\_top.do file we see the two files to simulate.



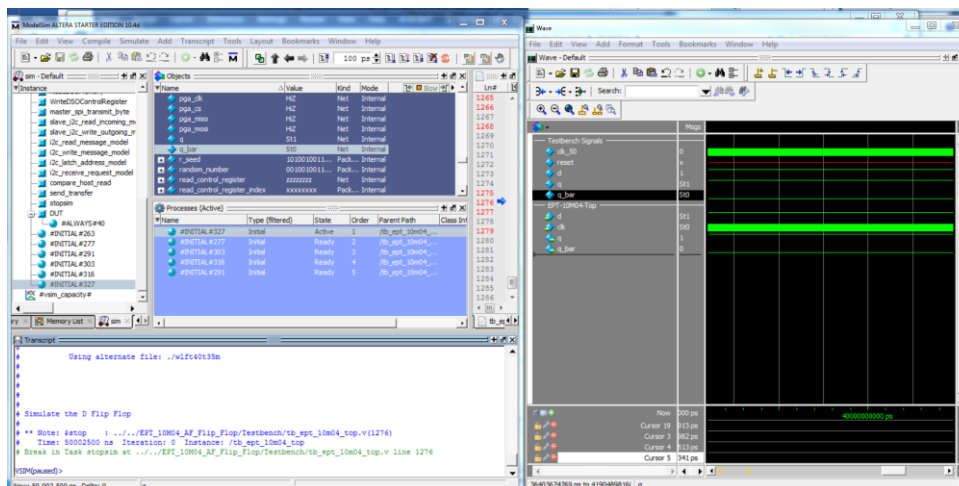
There are also simulate options that use the '+define' keyword. At the end of the file we see the 'do wave\_ept\_10m04\_top.do' instruction. This will add the display flags to the 'Waveforms' window.



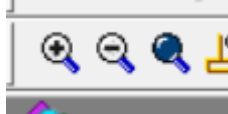
Start the simulation by typing `do sim_ept_10m04_top.do` into the command window.



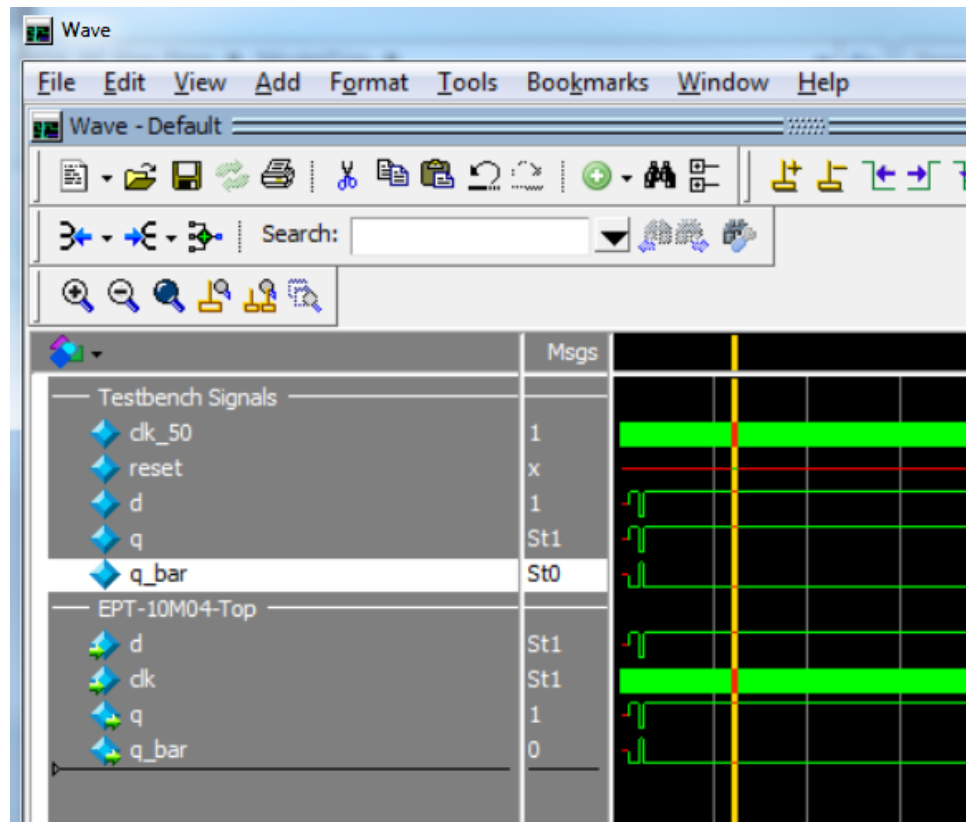
When the simulator completes its run, the Wave window appears.



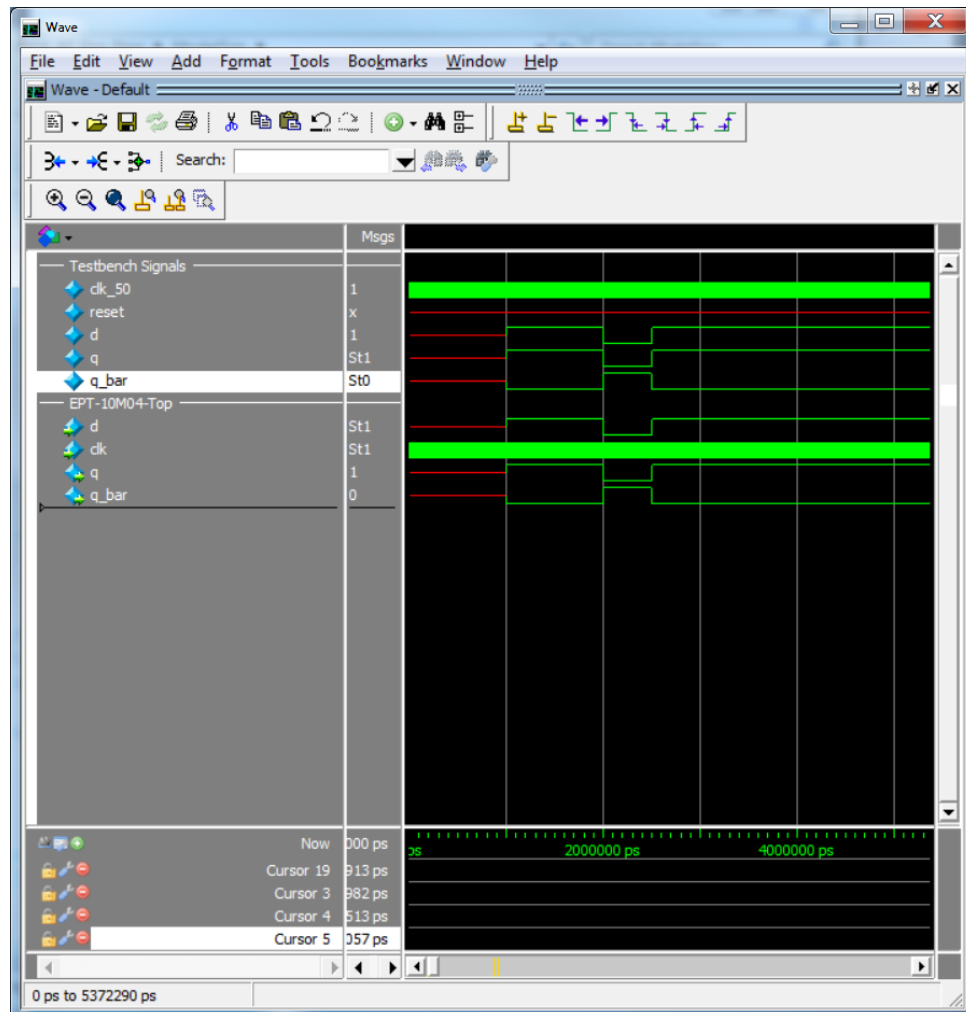
Zoom into the first 10 microseconds of the simulation using the Zoom



buttons.

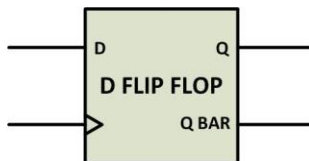


Now we see the 'd' stimulus toggling and the 'q' output following the input. Zoom in even farther.

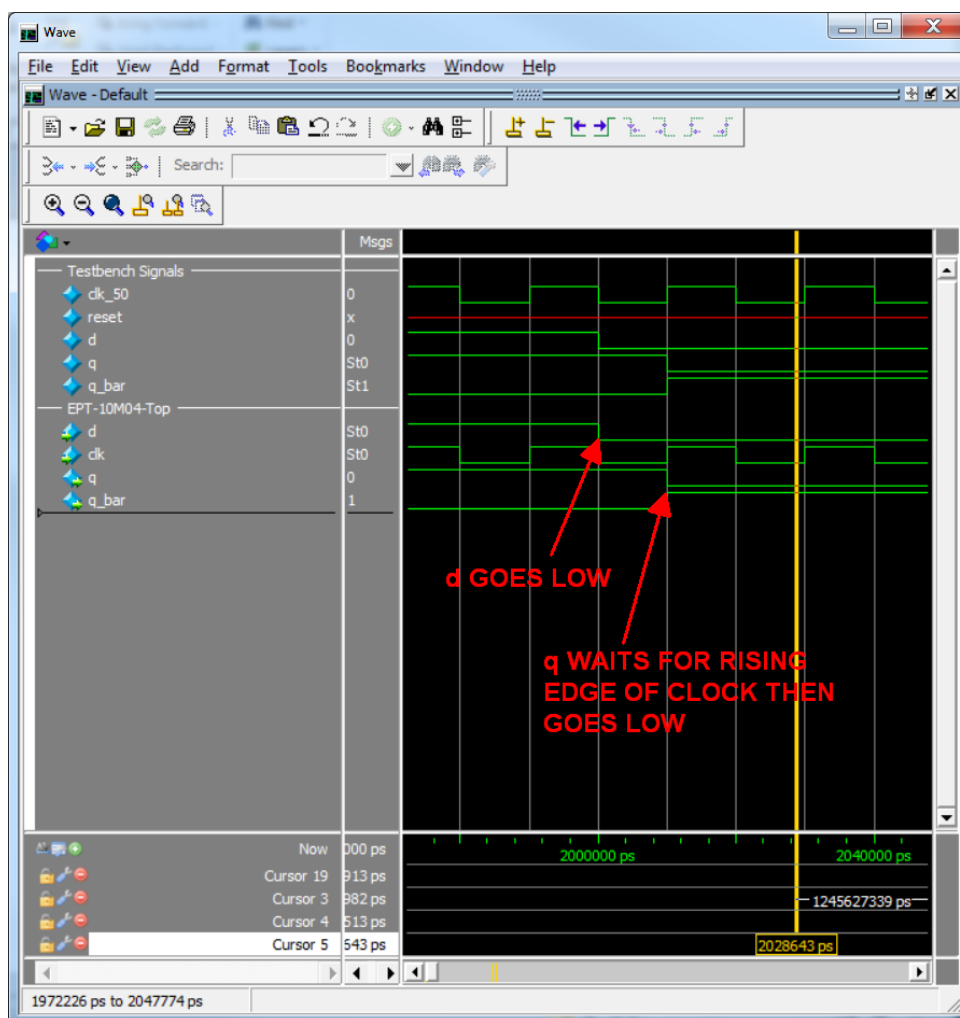


We can see after first delay that 'd' signal is asserted high. Before this assertion the signal is not defined in the simulation, so technically it is a Don't Care (indicated by the red 'floating' line). Once the 'd' is asserted, the delay is added by the simulator. After this delay the 'd' signal is de-asserted. Then, another delay and the 'd' is asserted high.

Now that we know what the simulator is doing, we can exam the user code, the D Flip Flop.



The D Flip Flop provides a registered output. This means that the input 'd' will be applied to the output 'q' but only after one rising edge of the clock. Because the output 'q' is synchronous to the clock, it is called a synchronous register. We can really see what this means when we zoom in even closer in the simulation.



Examining the user code, we can see this synchronous behavior occurring because of the always statement.

```
always @ (posedge clk)
begin
    q <= d;
    q_bar <= ! d;
end
```

The always keyword is used to cause a process to occur when an event happens. The event is what is in the paranthesis. In this case, the event is the rising edge of the 'clk' signal). Verilog uses the 'posedge' keyword to describe rising edge and 'clk' is our input clock from the testbench. So, what the Verilog code is telling us is that whenever we get a rising edge on the clock, the output 'q' is equal to 'd'. Also the 'q\_bar' output is set to the complement of 'd'. This is exactly what the simluation is showing us.

We will cover the details of the Verilog keywords, description of synchronous code and combinatorial code later in this tutorial. This first lesson was designed to get you started with using ModelSim.

## RTL DESCRIPTION

Many engineers who want to learn this language, very often ask this question, how much time will it take to learn Verilog? Well my answer to them is **"It may take no more than one week, if you happen to know at least one programming language"**.

### Design Styles

Verilog, like any other hardware description language, permits a design in either Bottom-up or Top-down methodology.

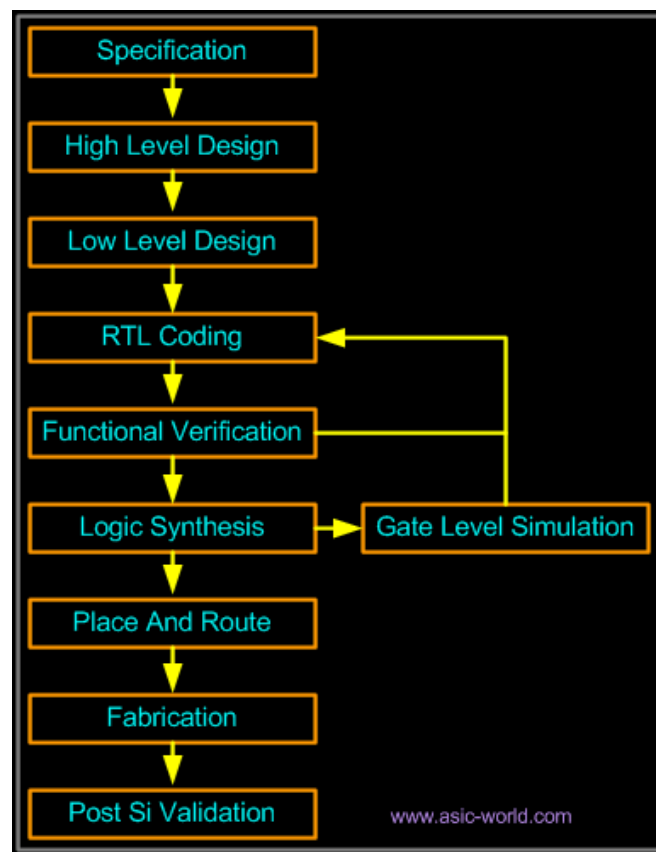
### Bottom-Up Design

The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standard gates (refer to the Digital Section for more details). With the increasing complexity of new designs this approach is nearly impossible to maintain. New systems consist of ASIC or microprocessors with a complexity of thousands of transistors. These traditional bottom-up designs have to give way to new structural, hierarchical design methods. Without these new practices it would be impossible to handle the new complexity.

### Top-Down Design

The desired design-style of all designers is the top-down one. A real top-down design allows early testing, easy change of different technologies, a structured system design and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact most designs are a mix of both methods, implementing some key elements of both design styles.

**Figure shows a Top-Down design approach.**



### Verilog Abstraction Levels

Verilog supports designing at many different levels of abstraction. Three of them are very important:

- Behavioral level
- Register-Transfer Level
- Gate Level

**Behavioral level**

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

**Register-Transfer Level**

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

**Gate Level**

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z'). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). *Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.*

## MODULES, PORTS, DATA TYPES AND OPERATORS

Every new learner's dream is to understand Verilog in one day, at least enough to use it. The next few pages are my attempt to make this dream a reality. There will be some theory and examples followed by some exercises. This tutorial will not teach you how to program; it is designed for those with some programming experience. Even though Verilog executes different code blocks concurrently as opposed to the sequential execution of most programming languages, there are still many parallels. Some background in digital design is also helpful.

Life before Verilog was a life full of schematics. Every design, regardless of complexity, was designed through schematics. They were difficult to verify and error-prone, resulting in long, tedious development cycles of design, verification... design, verification... design, verification...

When Verilog arrived, we suddenly had a different way of thinking about logic circuits. The Verilog design cycle is more like a traditional programming one, and it is what this tutorial will walk you through. Here's how it goes:

- Specifications (specs)
- High level design
- Low level (micro) design
- RTL coding
- Verification
- Synthesis.

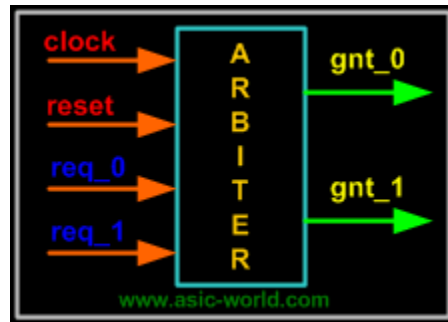
First on the list is **specifications** - what are the restrictions and requirements we will place on our design? What are we trying to build? For this tutorial, we'll be building a two agent arbiter: a device that selects among two agents competing for mastership. Here are some specs we might write up.

- Two agent arbiter.
- Active high asynchronous reset.
- Fixed priority, with agent 0 having priority over agent 1
- Grant will be asserted as long as request is asserted.

Once we have the specs, we can draw the block diagram, which is basically an abstraction of the data flow through a system (what goes into or comes out of the black boxes?). Since the example that we have taken is a simple one, we can have a block diagram as shown below. We don't worry about what's inside the magical black boxes just yet.

### Block diagram of arbiter

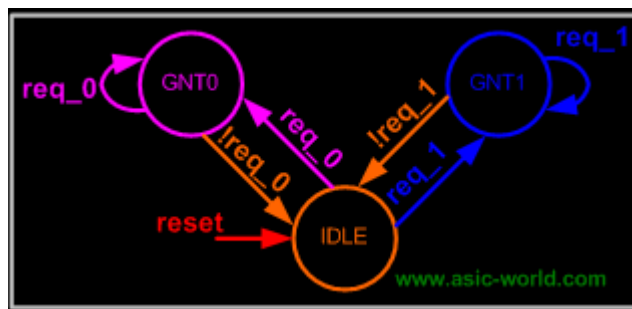




Now, if we were designing this machine without Verilog, the standard procedure would dictate that we draw a state machine. From there, we'd make a truth table with state transitions for each flip-flop. And after that we'd draw Karnaugh maps, and from K-maps we could get the optimized circuit. This method works just fine for small designs, but with large designs this flow becomes complicated and error prone. This is where Verilog comes in and shows us another way.

### Low level design

To see how Verilog helps us design our arbiter, let's go on to our state machine - now we're getting into the low-level design and peeling away the cover of the previous diagram's black box to see how our inputs affect the machine.



Each of the circles represents a **state** that the machine can be in. Each state corresponds to an output. The arrows between the states are state transitions, labeled by the event that causes the transition. For instance, the leftmost orange arrow means that if the machine is in state GNT0 (outputting the signal that corresponds to GNT0) and receives an input of !req\_0, the machine moves to state IDLE and outputs the signal that corresponds to that. This state machine describes all the logic of the system that you'll need. The next step is to put it all in Verilog.

## Modules

We'll need to backtrack a bit to do this. If you look at the arbiter block in the first picture, we can see that it has got a name ("arbiter") and input/output ports (req\_0, req\_1, gnt\_0, and gnt\_1).

Since Verilog is a HDL (hardware description language - one used for the conceptual design of integrated circuits), it also needs to have these things. In Verilog, we call our "black boxes" **module**. This is a reserved word within the program used to refer to things with inputs, outputs, and internal logic workings; they're the rough equivalents of functions with returns in other programming languages.

## Code of module "arbiter"

If you look closely at the arbiter block we see that there are arrow marks, (incoming for inputs and outgoing for outputs). In Verilog, after we have declared the module name and port names, we can define the direction of each port. (version note: In Verilog 2001 we can define ports and port directions at the same time) The code for this is shown below.

```
1 module arbiter (  
2 // Two slashes make a comment line.  
3 clock      , // clock  
4 reset      , // Active high, syn reset  
5 req_0      , // Request 0  
6 req_1      , // Request 1  
7 gnt_0      , // Grant 0  
8 gnt_1      // Grant 1  
9 );  
10 //-----Input Ports-----  
11 // Note : all commands are semicolon-delimited  
12 input      clock      ;  
13 input      reset      ;  
14 input      req_0      ;  
15 input      req_1      ;  
16 //-----Output Ports-----  
17 output     gnt_0      ;  
18 output     gnt_1      ;
```

You could download file one\_day1.v [here](#)

Here we have only two types of ports, input and output. In real life, we can have bi-directional ports as well. Verilog allows us to define bi-directional ports as "inout."

### Bi-Directional Ports Example -

```
inout read_enable; // port named read_enable is bi-directional
```

How do you define vector signals (signals composed of sequences of more than one bit)? Verilog provides a simple way to define these as well.

### Vector Signals Example -

```
inout [7:0] address; //port "address" is bidirectional
```

Note the [7:0] means we're using the little-endian convention - you start with 0 at the rightmost bit to begin the vector, then move to the left. If we had done [0:7], we would be using the big-endian convention and moving from left to right. Endianness is a purely arbitrary way of deciding which way your data will "read," but does differ between systems, so using the right endianness consistently is important. As an analogy, think of some languages (English) that are written left-to-right (big-endian) versus others (Arabic) written right-to-left (little-endian). Knowing which way the language flows is crucial to being able to read it, but the direction of flow itself was arbitrarily set years back.

### Summary

- We learnt how a block/module is defined in Verilog.
- We learnt how to define ports and port directions.
- We learnt how to declare vector/scalar ports.

### Data Type

What do data types have to do with hardware? Nothing, actually. People just wanted to write one more language that had data types in it. It's completely gratuitous; there's no point.

But wait... hardware does have two kinds of drivers.

(Drivers? What are those?)

A **driver** is a data type which can drive a load. Basically, in a physical circuit, a driver would be anything that electrons can move through/into.

- Driver that can store a value (example: flip-flop).
- Driver that can not store value, but connects two points (example: wire).

The first type of driver is called a reg in Verilog (short for "register"). The second data type is called a wire (for... well, "wire"). You can refer to tidbits section to understand it better.

There are lots of other data types - for instance, registers can be signed, unsigned, floating point... as a newbie, don't worry about them right now.

### Examples :

```
wire and_gate_output; // "and_gate_output" is a wire that only
                        outputs
reg d_flip_flop_output; // "d_flip_flop_output" is a register; it stores
                        and outputs a value
reg [7:0] address_bus; // "address_bus" is a little-endian 8-bit
                        register
```

### Summary

- Wire data type is used for connecting two points.
- Reg data type is used for storing values.
- May god bless the rest of data types. You'll see them someday.

### Operators

Operators, thankfully, are the same things here as they are in other programming languages. They take two values and compare (or otherwise operate on) them to yield a third result - common examples are addition, equals, logical-and... To make life easier for us, nearly all operators (at least the ones in the list below) are exactly the same as their counterparts in the C programming language.

| Operator Type        | Operator Symbol | Operation Performed   |
|----------------------|-----------------|-----------------------|
| <b>Arithmetic</b>    | *               | Multiply              |
|                      | /               | Division              |
|                      | +               | Add                   |
|                      | -               | Subtract              |
|                      | %               | Modulus               |
|                      | +               | Unary plus            |
|                      | -               | Unary minus           |
| <b>Logical</b>       | !               | Logical negation      |
|                      | &&              | Logical and           |
|                      |                 | Logical or            |
| <b>Relational</b>    | >               | Greater than          |
|                      | <               | Less than             |
|                      | >=              | Greater than or equal |
|                      | <=              | Less than or equal    |
| <b>Equality</b>      | ==              | Equality              |
|                      | !=              | inequality            |
| <b>Reduction</b>     | ~               | Bitwise negation      |
|                      | ~&              | nand                  |
|                      |                 | or                    |
|                      | ~               | nor                   |
|                      | ^               | xor                   |
|                      | ^~              | xnor                  |
| <b>Shift</b>         | >>              | Right shift           |
|                      | <<              | Left shift            |
| <b>Concatenation</b> | { }             | Concatenation         |
| <b>Conditional</b>   | ?               | conditional           |

////////////////////////////////Lesson #2////////////////////////////////

In this first lesson, we will use some operators in Verilog code.

First, lets write some user code exercise the operators.

### Example -

- `a = b + c ; // That was very easy`
- `a = 1 << 5; // Hum let me think, ok shift '1' left by 5 positions.`
- `a = !b ; // Well does it invert b???`
- `a = ~b ; // How many times do you want to assign to 'a', it could cause multiple-drivers.`

### Summary

- Let's attend the C language training again, they're (almost) just like the C ones.

## CONTROL STATEMENTS

Wait, what's this? **if, else, repeat, while, for, case** - it's Verilog that looks exactly like C (and probably whatever other language you're used to program in)! Even though the functionality appears to be the same as in C, Verilog is an HDL, so the descriptions should translate to hardware. This means you've got to be careful when using control statements (otherwise your designs might not be implementable in hardware).

### If-else

If-else statements check a condition to decide whether or not to execute a portion of code. If a condition is satisfied, the code is executed. Else, it runs this other portion of code.

```
1 // begin and end act like curly braces in C/C++.
2 if (enable == 1'b1) begin
3     data = 10; // Decimal assigned
4     address = 16'hDEAD; // Hexadecimal
5     wr_enable = 1'b1; // Binary
6 end else begin
7     data = 32'b0;
8     wr_enable = 1'b0;
9     address = address + 1;
```

```
10 end
```

You could download file one\_day2.v [here](#)

One could use any operator in the condition checking, as in the case of C language. If needed we can have nested if else statements; statements without else are also ok, but they have their own problem, when modeling combinational logic, in case they result in a Latch (this is not always true).

### Case

Case statements are used where we have one variable which needs to be checked for multiple values. like an address decoder, where the input is an address and it needs to be checked for all the values that it can take. Instead of using multiple nested if-else statements, one for each value we're looking for, we use a single case statement: this is similar to switch statements in languages like C++.

Case statements begin with the reserved word **case** and end with the reserved word **endcase** (Verilog does not use brackets to delimit blocks of code). The cases, followed with a colon and the statements you wish executed, are listed within these two delimiters. It's also a good idea to have a **default** case. Just like with a finite state machine (FSM), if the Verilog machine enters into a non-covered statement, the machine hangs. Defaulting the statement with a return to idle keeps us safe.

```
1 case (address)
2   0 : $display ("It is 11:40PM") ;
3   1 : $display ("I am feeling sleepy") ;
4   2 : $display ("Let me skip this tutorial") ;
5   default : $display ("Need to complete") ;
6 endcase
```

You could download file one\_day3.v [here](#)

Looks like the address value was 3 and so I am still writing this tutorial.

**Note:** One thing that is common to if-else and case statement is that, if you don't cover all the cases (don't have 'else' in If-else or 'default' in Case), and you are trying to write a combinational statement, the synthesis tool will infer Latch.

### While

A while statement executes the code within it repeatedly if the condition it is assigned to check returns true. While loops are not normally used for models in real life, but they are used in test benches. As with other statement blocks, they are delimited by begin and end.

```
1 while (free_time) begin
2   $display ("Continue with webpage development");
3 end
```

You could download file one\_day4.v [here](#)

As long as free\_time variable is set, code within the begin and end will be executed. i.e print "Continue with web development". Let's look at a stranger example, which uses most of Verilog constructs. Well, you heard it right. Verilog has fewer reserved words than VHDL, and in this few, we use even lesser for actual coding. So good of Verilog... so right.

```
1 module counter (clk,rst,enable,count);
2 input clk, rst, enable;
3 output [3:0] count;
4 reg [3:0] count;
5
6 always @ (posedge clk or posedge rst)
7 if (rst) begin
8   count <= 0;
9 end else begin : COUNT
10   while (enable) begin
11     count <= count + 1;
12     disable COUNT;
13   end
14 end
15
16 endmodule
```

You could download file one\_day5.v [here](#)

The example above uses most of the constructs of Verilog. You'll notice a new block called always - this illustrates one of the key features of Verilog. Most software languages, as we mentioned before, execute sequentially - that is, statement by statement. Verilog programs, on the other hand, often have many statements executing in parallel. All blocks marked always will run - simultaneously - when one or more of the conditions listed within it is fulfilled.



In the example above, the always block will run when either rst or clk reaches a **positive edge** - that is, when their value has risen from 0 to 1. You can have two or more **always** blocks in a program going at the same time (not shown here, but commonly used).

We can disable a block of code, by using the reserve word disable. In the above example, after each counter increment, the COUNT block of code (not shown here) is disabled.

### For loop

For loops in Verilog are almost exactly like for loops in C or C++. The only difference is that the ++ and -- operators are not supported in Verilog. Instead of writing i++ as you would in C, you need to write out its full operational equivalent, i = i + 1.

```
1   for (i = 0; i < 16; i = i + 1) begin
2       $display ("Current value of i is %d", i);
3   end
```

You could download file one\_day6.v [here](#)

This code will print the numbers from 0 to 15 in order. Be careful when using for loops for register transfer logic (RTL) and make sure your code is actually sanely implementable in hardware... and that your loop is not infinite.

### Repeat

Repeat is similar to the for loop we just covered. Instead of explicitly specifying a variable and incrementing it when we declare the for loop, we tell the program how many times to run through the code, and no variables are incremented (unless we want them to be, like in this example).

```
1   repeat (16) begin
2       $display ("Current value of i is %d", i);
3       i = i + 1;
4   end
```

You could download file one\_day7.v [here](#)

The output is exactly the same as in the previous for-loop program example. It is relatively rare to use a repeat (or for-loop) in actual hardware implementation.

### Summary

- While, if-else, case(switch) statements are the same as in C language.
- If-else and case statements require all the cases to be covered for combinational logic.
- For-loop is the same as in C, but no ++ and -- operators.
- Repeat is the same as the for-loop but without the incrementing variable.

### Variable Assignment

In digital there are two types of elements, combinational and sequential. Of course we know this. But the question is "How do we model this in Verilog?". Well Verilog provides two ways to model the combinational logic and only one way to model sequential logic.

- Combinational elements can be modeled using assign and always statements.
- Sequential elements can be modeled using only always statement.
- There is a third block, which is used in test benches only: it is called Initial statement.

### Initial Blocks

An initial block, as the name suggests, is executed only once when simulation starts. This is useful in writing test benches. If we have multiple initial blocks, then all of them are executed at the beginning of simulation.

### Example

```
1 initial begin
2     clk = 0;
3     reset = 0;
4     req_0 = 0;
5     req_1 = 0;
6 end
```

You could download file one\_day8.v [here](#)

In the above example, at the beginning of simulation, (i.e. when time = 0), all the variables inside the begin and end block are driven zero.

## Always Blocks

As the name suggests, an always block executes always, unlike initial blocks which execute only once (at the beginning of simulation). A second difference is that an always block should have a sensitive list or a delay associated with it.

The sensitive list is the one which tells the always block when to execute the block of code, as shown in the figure below. The @ symbol after reserved word 'always', indicates that the block will be triggered "at" the condition in parenthesis after symbol @.

One important note about always block: it can not drive wire data type, but can drive reg and integer data types.

```
1 always @ (a or b or sel)
2 begin
3     y = 0;
4     if (sel == 0) begin
5         y = a;
6     end else begin
7         y = b;
8     end
9 end
```

You could download file one\_day9.v [here](#)

The above example is a 2:1 mux, with input a and b; sel is the select input and y is the mux output. In any combinational logic, output changes whenever input changes. This theory when applied to always blocks means that the code inside always blocks needs to be executed whenever the input variables (or output controlling variables) change. These variables are the ones included in the sensitive list, namely a, b and sel.

There are two types of sensitive list: level sensitive (for combinational circuits) and edge sensitive (for flip-flops). The code below is the same 2:1 Mux but the output y is now a flip-flop output.

```
1 always @ (posedge clk )
2 if (reset == 0) begin
3     y <= 0;
4 end else if (sel == 0) begin
5     y <= a;
6 end else begin
7     y <= b;
8 end
```

You could download file one\_day10.v [here](#)

We normally have to reset flip-flops, thus every time the clock makes the transition from 0 to 1 (posedge), we check if reset is asserted (synchronous reset), then we go on with normal logic. If we look closely we see that in the case of combinational logic we had "=" for assignment, and for the sequential block we had the "<=" operator. Well, "=" is blocking assignment and "<=" is nonblocking assignment. "=" executes code sequentially inside a begin / end, whereas nonblocking "<=" executes in parallel.

We can have an always block without sensitive list, in this case we need to have a delay as shown in the code below.

```
1 always begin
2     #5 clk = ~clk;
3 end
```

You could download file one\_day11.v [here](#)

#5 in front of the statement delays its execution by 5 time units.

## Assign Statement

An assign statement is used for modeling only combinational logic and it is executed continuously. So the assign statement is called 'continuous assignment statement' as there is no sensitive list.

```
1 assign out = (enable) ? data : 1'bz;
```

You could download file one\_day12.v [here](#)

The above example is a tri-state buffer. When enable is 1, data is driven to out, else out is pulled to high-impedance. We can have nested conditional operators to construct mux, decoders and encoders.

```
1 assign out = data;
```

You could download file one\_day13.v [here](#)

This example is a simple buffer.

### Task and Function

When repeating the same old things again and again, Verilog, like any other programming language, provides means to address repeated used code, these are called Tasks and Functions. I wish I had something similar for webpages, just call it to print this programming language stuff again and again.

Code below is used for calculating even parity.

```
1 function parity;
2 input [31:0] data;
3 integer i;
4 begin
5     parity = 0;
6     for (i= 0; i < 32; i = i + 1) begin
7         parity = parity ^ data[i];
8     end
9 end
10 endfunction
```

You could download file one\_day14.v [here](#)

Functions and tasks have the same syntax; one difference is that tasks can have delays, whereas functions can not have any delay. This means that function can be used for modeling combinational logic.

A second difference is that functions can return a value, whereas tasks can not.

## TEST BENCHES

Ok, we have code written according to the design document, now what?

Well we need to test it to see if it works according to specs. Most of the time, it's the same we use to do in digital labs in college days: drive

the inputs, match the outputs with expected values. Let's look at the arbiter testbench.

```
1 module arbiter (  
2   clock,  
3   reset,  
4   req_0,  
5   req_1,  
6   gnt_0,  
7   gnt_1  
8 );  
9  
10 input clock, reset, req_0, req_1;  
11 output gnt_0, gnt_1;  
12  
13 reg gnt_0, gnt_1;  
14  
15 always @ (posedge clock or posedge reset)  
16 if (reset) begin  
17   gnt_0 <= 0;  
18   gnt_1 <= 0;  
19 end else if (req_0) begin  
20   gnt_0 <= 1;  
21   gnt_1 <= 0;  
22 end else if (req_1) begin  
23   gnt_0 <= 0;  
24   gnt_1 <= 1;  
25 end  
26  
27 endmodule  
28 // Testbench Code Goes here  
29 module arbiter_tb;  
30  
31 reg clock, reset, req0, req1;  
32 wire gnt0, gnt1;  
33  
34 initial begin  
35   $monitor ("req0=%b, req1=%b, gnt0=%b, gnt1=%b",  
36 req0, req1, gnt0, gnt1);  
37   clock = 0;  
38   reset = 0;  
39   req0 = 0;  
40   req1 = 0;  
41   #5 reset = 1;  
42   #15 reset = 0;  
43   #10 req0 = 1;  
44   #10 req0 = 0;  
45   #10 req1 = 1;
```

```

45    #10 req1 = 0;
46    #10 {req0, req1} = 2'b11;
47    #10 {req0, req1} = 2'b00;
48    #10 $finish;
49 end
50
51 always begin
52    #5 clock = !clock;
53 end
54
55 arbiter U0 (
56    .clock (clock),
57    .reset (reset),
58    .req_0 (req0),
59    .req_1 (req1),
60    .gnt_0 (gnt0),
61    .gnt_1 (gnt1)
62 );
63
64 endmodule

```

You could download file arbiter.v [here](#)

It looks like we have declared all the arbiter inputs as reg and outputs as wire; well, that's true. We are doing this as test bench needs to drive inputs and needs to monitor outputs.

After we have declared all needed variables, we initialize all the inputs to known state: we do that in the initial block. After initialization, we assert/de-assert reset, req0, req1 in the sequence we want to test the arbiter. Clock is generated with an always block.

After we are done with the testing, we need to stop the simulator. Well, we use \$finish to terminate simulation. \$monitor is used to monitor the changes in the signal list and print them in the format we want.

```

req0=0,req1=0,gnt0=x,gnt1=x
req0=0,req1=0,gnt0=0,gnt1=0
req0=1,req1=0,gnt0=0,gnt1=0
req0=1,req1=0,gnt0=1,gnt1=0
req0=0,req1=0,gnt0=1,gnt1=0
req0=0,req1=1,gnt0=1,gnt1=0
req0=0,req1=1,gnt0=0,gnt1=1
req0=0,req1=0,gnt0=0,gnt1=1
req0=1,req1=1,gnt0=0,gnt1=1
req0=1,req1=1,gnt0=1,gnt1=0
req0=0,req1=0,gnt0=1,gnt1=0

```

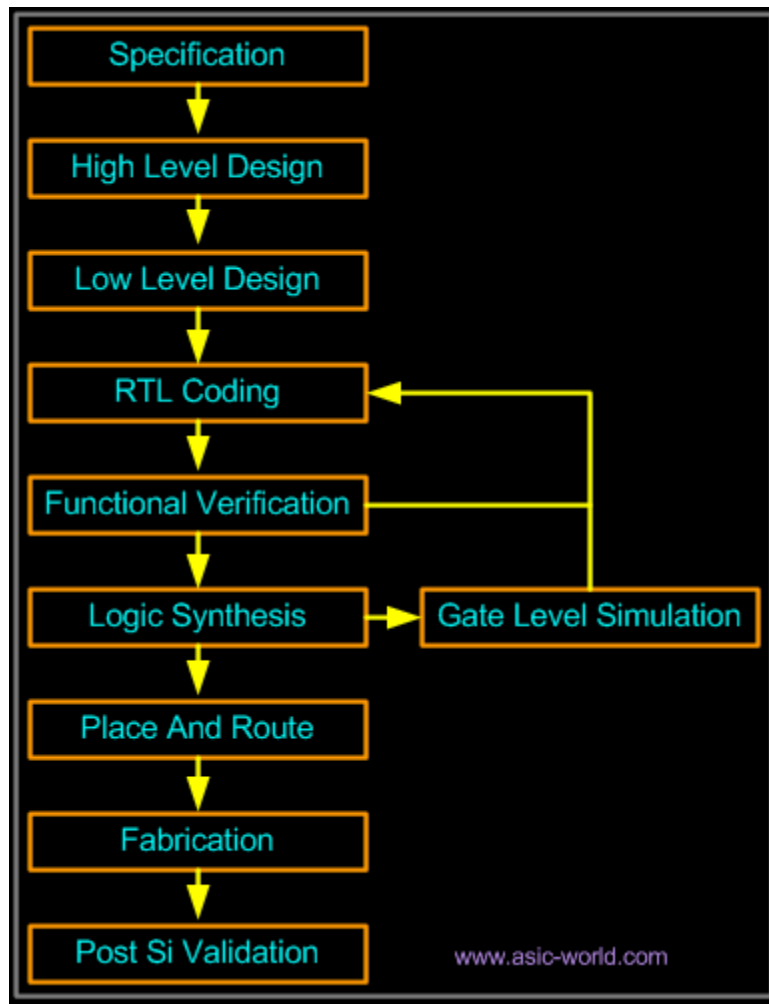
Being new to Verilog you might want to try some examples and try designing something new. I have listed the tool flow that could be used to achieve this. I have personally tried this flow and found this to be working just fine for me. Here I have taken only the front end design part and bits of FPGA design of the tool flow, that can be done without any fat money spent on tools.

### ● Various stages of ASIC/FPGA

- **Specification** : Word processor like Word, Kwriter, AbiWord, Open Office.
- **High Level Design** : Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word, Open Office.
- **Micro Design/Low level design**: Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word.
- **RTL Coding** : Vim, Emacs, conTEXT, HDL TurboWriter
- **Simulation** : Modelsim, VCS, Verilog-XL, Veriwell, Finsim, Icarus.
- **Synthesis** : Design Compiler, FPGA Compiler, Synplify, Leonardo Spectrum. You can download this from FPGA vendors like Altera and Xilinx for free.
- **Place & Route** : For FPGA use FPGA' vendors P&R tool. ASIC tools require expensive P&R tools like Apollo. Students can use LASI, Magic.
- **Post Si Validation** : For ASIC and FPGA, the chip needs to be tested in real environment. Board design, device drivers needs to be in place.

### ◆ Figure : Typical Design flow





### ◆ Specification

This is the stage at which we define what are the important parameters of the system/design that you are planning to design. A simple example would be: I want to design a counter; it should be 4 bit wide, should have synchronous reset, with active high enable; when reset is active, counter output should go to "0".

### ◆ High Level Design

This is the stage at which you define various blocks in the design and how they communicate. Let's assume that we need to design a microprocessor: high level design means splitting the design into blocks based on their function; in our case the blocks are registers, ALU, Instruction Decode, Memory Interface, etc.

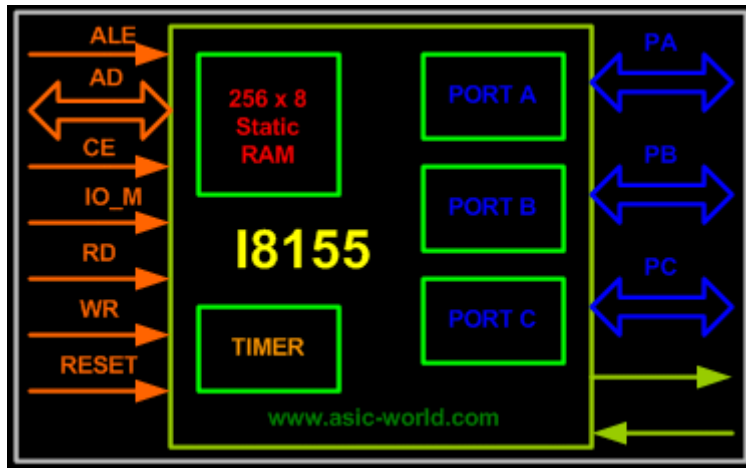


Figure : I8155 High Level Block Diagram

### Micro Design/Low level design

Low level design or Micro design is the phase in which the designer describes how each block is implemented. It contains details of State machines, counters, Mux, decoders, internal registers. It is always a good idea to draw waveforms at various interfaces. This is the phase where one spends lot of time.

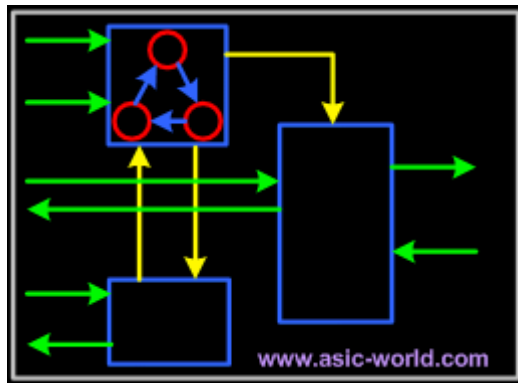


Figure : Sample Low level design

### RTL Coding

In RTL coding, Micro design is converted into Verilog/VHDL code, using synthesizable constructs of the language. Normally we like to lint the code, before starting verification or synthesis.

```
1 module addbit (
```

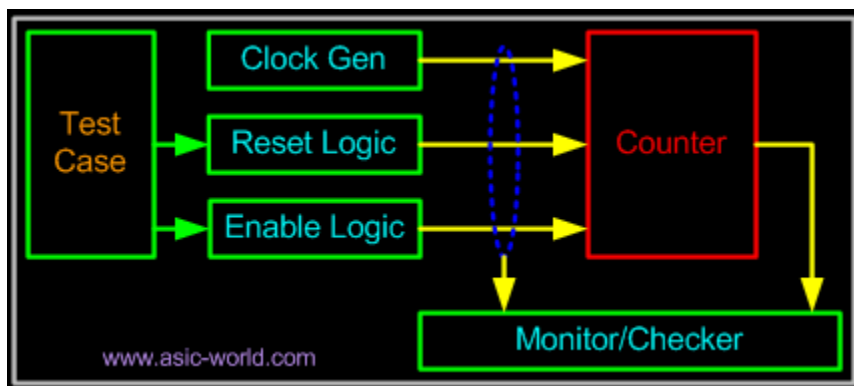
```

2  a      , // first input
3  b      , // Second input
4  ci     , // Carry input
5  sum    , // sum output
6  co     // carry output
7  );
8  //Input declaration
9  input a;
10 input b;
11 input ci;
12 //Output declaration
13 output sum;
14 output co;
15 //Port Data types
16 wire a;
17 wire b;
18 wire ci;
19 wire sum;
20 wire co;
21 //Code starts here
22 assign {co,sum} = a + b + ci;
23
24 endmodule // End of Module addbit
  
```

You could download file addbit.v [here](#)

### Simulation

Simulation is the process of verifying the functional characteristics of models at any level of abstraction. We use simulators to simulate the Hardware models. To test if the RTL code meets the functional requirements of the specification, we must see if all the RTL blocks are functionally correct. To achieve this we need to write a testbench, which generates clk, reset and the required test vectors. A sample testbench for a counter is shown below. Normally we spend 60-70% of time in design verification.



### Figure : Sample Testbench Env

We use the waveform output from the simulator to see if the DUT (Device Under Test) is functionally correct. Most of the simulators come with a waveform viewer. As design becomes complex, we write self checking testbench, where testbench applies the test vector, then compares the output of DUT with expected values.

There is another kind of simulation, called **timing simulation**, which is done after synthesis or after P&R (Place and Route). Here we include the gate delays and wire delays and see if DUT works at rated clock speed. This is also called as **SDF simulation** or **gate level simulation**.

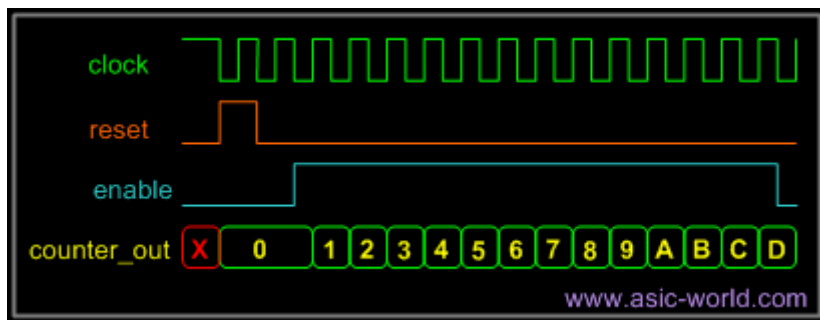
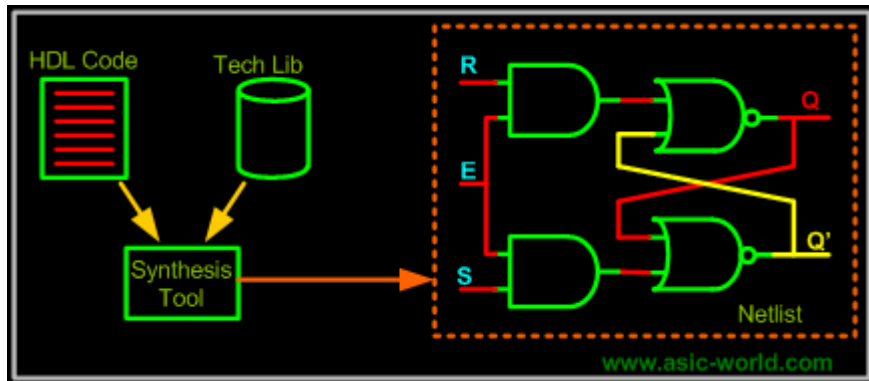


Figure : 4 bit Up Counter Waveform

### Synthesis

Synthesis is the process in which synthesis tools like design compiler or Synplify take RTL in Verilog or VHDL, target technology, and constraints as input and maps the RTL to target technology primitives. Synthesis tool, after mapping the RTL to gates, also do the minimal amount of timing analysis to see if the mapped design is meeting the timing requirements. (Important thing to note is, synthesis tools are not aware of wire delays, they only know of gate delays). After the synthesis there are a couple of things that are normally done before passing the netlist to backend (Place and Route)

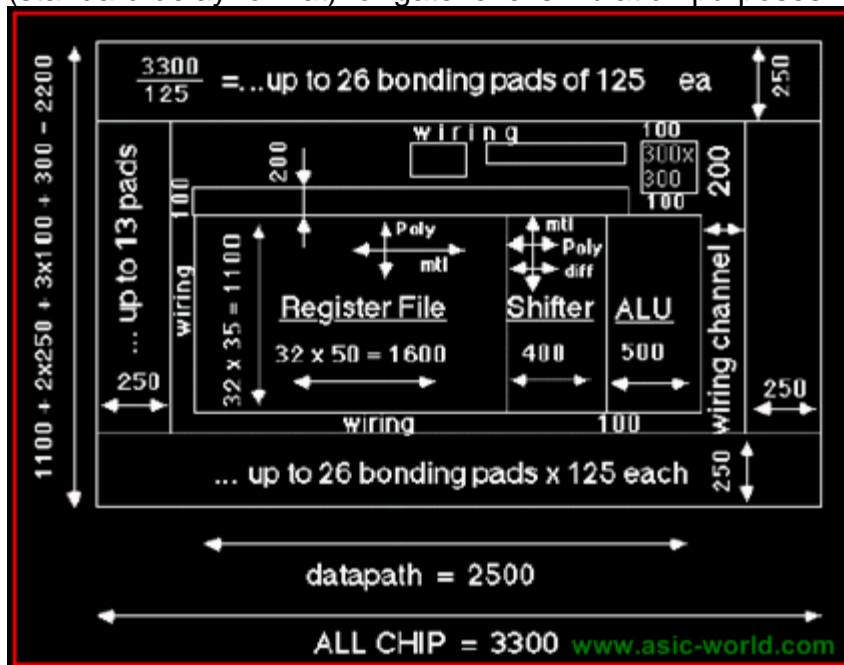
- **Formal Verification** : Check if the RTL to gate mapping is correct.
- **Scan insertion** : Insert the scan chain in the case of ASIC.



### Figure : Synthesis Flow

## Place & Route

The gatelevel netlist from the synthesis tool is taken and imported into place and route tool in Verilog netlist format. All the gates and flip-flops are placed; clock tree synthesis and reset is routed. After this each block is routed. The P&R tool output is a GDS file, used by foundry for fabricating the ASIC. Backend team normally dumps out SPEF (standard parasitic exchange format) /RSPF (reduced parasitic exchange format)/DSPF (detailed parasitic exchange format) from layout tools like ASTRO to the frontend team, who then use the read\_parasitic command in tools like Prime Time to write out SDF (standard delay format) for gate level simulation purposes.



**Figure : Sample micro-processor placement**

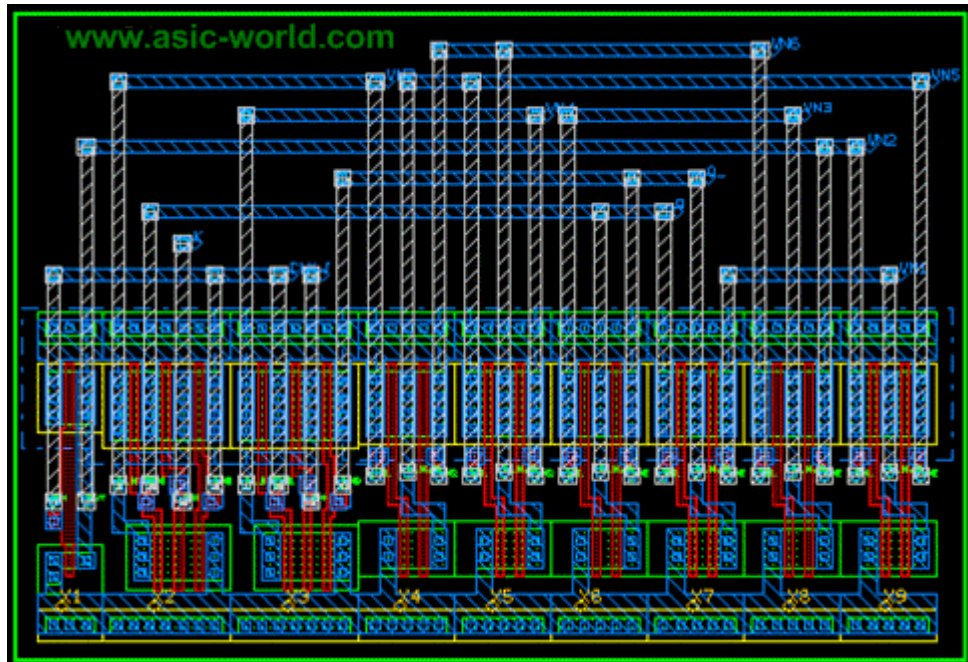


Figure : J-K Flip-Flop

### Post Silicon Validation

Once the chip (silicon) is back from fab, it needs to be put in a real environment and tested before it can be released into Market. Since the simulation speed (number of clocks per second) with RTL is very slow, there is always the possibility to find a bug in Post silicon validation.

### Introduction

If you refer to any book on programming languages, it starts with an "Hello World" program; once you have written it, you can be sure that you can do something in that language 😊.

Well I am also going to show how to write a **"hello world"** program, followed by a **"counter"** design, in Verilog.

### ● Hello World Program

```

1 //-----
2 // This is my first Verilog Program
3 // Design Name : hello_world
4 // File Name : hello_world.v
  
```

```

5 // Function : This program will print 'hello world'
6 // Coder   : Deepak
7 //-----
8 module hello_world ;
9
10 initial begin
11     $display ("Hello World by Deepak");
12     #10 $finish;
13 end
14
15 endmodule // End of Module hello_world
  
```

You could download file hello\_world.v [here](#)

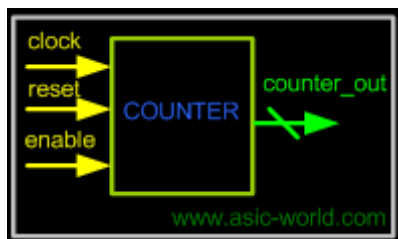
Words in green are comments, blue are reserved words. Any program in Verilog starts with reserved word 'module' <module\_name>. In the above example line 8 contains module hello\_world. (Note: We can have compiler pre-processor statements like `include', `define' before module declaration)

Line 10 contains the initial block: this block gets executed only once after the simulation starts, at time=0 (0ns). This block contains two statements which are enclosed within begin, at line 10, and end, at line 13. In Verilog, if you have multiple lines within a block, you need to use begin and end. Module ends with 'endmodule' reserved word, in this case at line 15.

### Hello World Program Output

Hello World by Deepak

### Counter Design Block



### Counter Design Specs

- 4-bit synchronous up counter.

- active high, synchronous reset.
- Active high enable.

## Counter Design

```

1 //-----
2 // This is my second Verilog Design
3 // Design Name : first_counter
4 // File Name : first_counter.v
5 // Function : This is a 4 bit up-counter with
6 // Synchronous active high reset and
7 // with active high enable signal
8 //-----
9 module first_counter (
10 clock , // Clock input of the design
11 reset , // active high, synchronous Reset input
12 enable , // Active high enable signal for counter
13 counter_out // 4 bit vector output of the counter
14 ); // End of port list
15 //-----Input Ports-----
16 input clock ;
17 input reset ;
18 input enable ;
19 //-----Output Ports-----
20 output [3:0] counter_out ;
21 //-----Input ports Data Type-----
22 // By rule all the input ports should be wires
23 wire clock ;
24 wire reset ;
25 wire enable ;
26 //-----Output Ports Data Type-----
27 // Output port can be a storage element (reg) or a wire
28 reg [3:0] counter_out ;
29
30 //-----Code Starts Here-----
31 // Since this counter is a positive edge triggered one,
32 // We trigger the below block with respect to positive
33 // edge of the clock.
34 always @ (posedge clock)
35 begin : COUNTER // Block Name
36   // At every rising edge of clock we check if reset is active
37   // If active, we load the counter output with 4'b0000
38   if (reset == 1'b1) begin
39     counter_out <= #1 4'b0000;
40   end

```



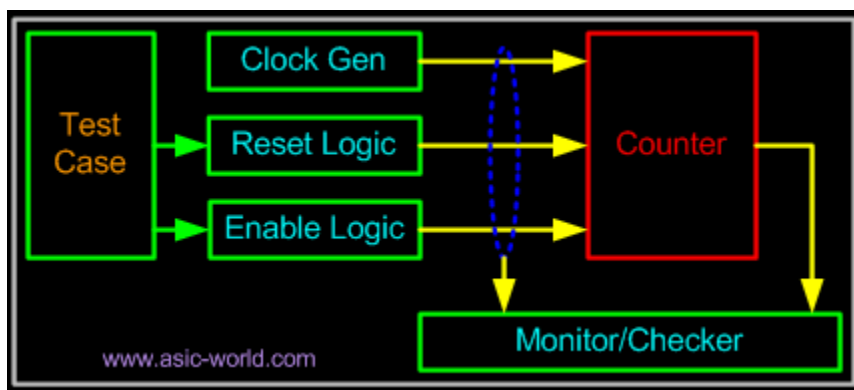
```

41 // If enable is active, then we increment the counter
42 else if (enable == 1'b1) begin
43     counter_out <= #1 counter_out + 1;
44 end
45 end // End of Block COUNTER
46
47 endmodule // End of Module counter
  
```

You could download file first\_counter.v [here](#)

### Counter Test Bench

Any digital circuit, no matter how complex, needs to be tested. For the counter logic, we need to provide clock and reset logic. Once the counter is out of reset, we toggle the enable input to the counter, and check the waveform to see if the counter is counting correctly. This is done in Verilog.



The counter testbench consists of clock generator, reset control, enable control and monitor/checker logic. Below is the simple code of testbench without the monitor/checker logic.

```

1 `include "first_counter.v"
2 module first_counter_tb();
3 // Declare inputs as regs and outputs as wires
4 reg clock, reset, enable;
5 wire [3:0] counter_out;
6
7 // Initialize all variables
8 initial begin
9     $display ("time\t clk reset enable counter");
10    $monitor ("%g\t %b %b %b %b",
11        $time, clock, reset, enable, counter_out);
12    clock = 1; // initial value of clock
13    reset = 0; // initial value of reset
  
```

```

14  enable = 0;           // initial value of enable
15  #5  reset = 1;        // Assert the reset
16  #10 reset = 0;        // De-assert the reset
17  #10 enable = 1;       // Assert enable
18  #100 enable = 0;      // De-assert enable
19  #5  $finish;          // Terminate simulation
20  end
21
22  // Clock generator
23  always begin
24    #5  clock = ~clock; // Toggle clock every 5 ticks
25  end
26
27  // Connect DUT to test bench
28  first_counter U_counter (
29    clock,
30    reset,
31    enable,
32    counter_out
33  );
34
35  endmodule

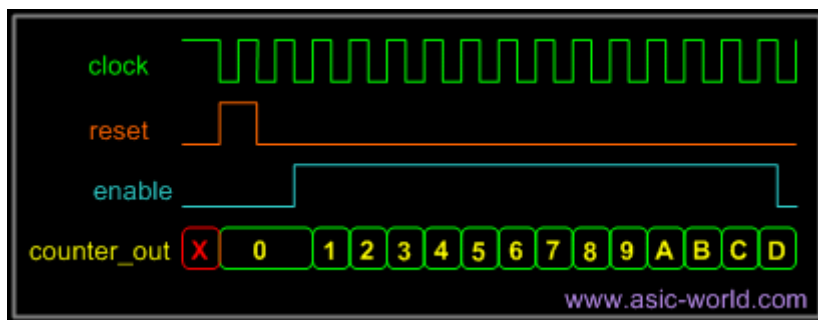
```

You could download file first\_counter\_tb.v [here](#)

| time | clk | reset | enable | counter |
|------|-----|-------|--------|---------|
| 0    | 1   | 0     | 0      | xxxx    |
| 5    | 0   | 1     | 0      | xxxx    |
| 10   | 1   | 1     | 0      | xxxx    |
| 11   | 1   | 1     | 0      | 0000    |
| 15   | 0   | 0     | 0      | 0000    |
| 20   | 1   | 0     | 0      | 0000    |
| 25   | 0   | 0     | 1      | 0000    |
| 30   | 1   | 0     | 1      | 0000    |
| 31   | 1   | 0     | 1      | 0001    |
| 35   | 0   | 0     | 1      | 0001    |
| 40   | 1   | 0     | 1      | 0001    |
| 41   | 1   | 0     | 1      | 0010    |
| 45   | 0   | 0     | 1      | 0010    |
| 50   | 1   | 0     | 1      | 0010    |
| 51   | 1   | 0     | 1      | 0011    |
| 55   | 0   | 0     | 1      | 0011    |
| 60   | 1   | 0     | 1      | 0011    |
| 61   | 1   | 0     | 1      | 0100    |
| 65   | 0   | 0     | 1      | 0100    |
| 70   | 1   | 0     | 1      | 0100    |
| 71   | 1   | 0     | 1      | 0101    |
| 75   | 0   | 0     | 1      | 0101    |
| 80   | 1   | 0     | 1      | 0101    |
| 81   | 1   | 0     | 1      | 0110    |
| 85   | 0   | 0     | 1      | 0110    |

|     |   |   |   |      |
|-----|---|---|---|------|
| 90  | 1 | 0 | 1 | 0110 |
| 91  | 1 | 0 | 1 | 0111 |
| 95  | 0 | 0 | 1 | 0111 |
| 100 | 1 | 0 | 1 | 0111 |
| 101 | 1 | 0 | 1 | 1000 |
| 105 | 0 | 0 | 1 | 1000 |
| 110 | 1 | 0 | 1 | 1000 |
| 111 | 1 | 0 | 1 | 1001 |
| 115 | 0 | 0 | 1 | 1001 |
| 120 | 1 | 0 | 1 | 1001 |
| 121 | 1 | 0 | 1 | 1010 |
| 125 | 0 | 0 | 0 | 1010 |

### Counter Waveform



## Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

### White Space

White space can contain the characters for blanks, tabs, newlines, and form feeds. These characters are ignored except when they serve to separate other tokens. However, blanks and tabs are significant in strings.

White space characters are :

- Blank spaces
- Tabs
- Carriage returns
- New-line
- Form-feeds

## ✦ Examples of White Spaces

### Functional Equivalent Code

**Bad Code** : Never write code like this.

```
1 module addbit(a,b,ci,sum,co) ;
2 input a,b,ci;output sum co;
3 wire a,b,ci,sum,co;endmodule
```

You could download file bad\_code.v [here](#)

**Good Code** : Nice way to write code.

```
1      module addbit (
2          a,
3          b,
4          ci,
5          sum,
6          co) ;
7      input      a;
8      input      b;
9      input      ci;
10     output      sum;
11     output      co;
12     wire      a;
13     wire      b;
14     wire      ci;
15     wire      sum;
16     wire      co;
17
18     endmodule
```

You could download file good\_code.v [here](#)

## ✦ Comments

There are two forms to introduce comments.

- Single line comments begin with the token `//` and end with a carriage return
- Multi line comments begin with the token `/*` and end with the token `*/`

## ✦ Examples of Comments

```
1 /* This is a
2    Multi line comment
3    example */
4 module addbit (
5     a,
6     b,
7     ci,
8     sum,
9     co);
10
11 // Input Ports Single line comment
12 input      a;
13 input      b;
14 input      ci;
15 // Output ports
16 output     sum;
17 output     co;
18 // Data Types
19 wire       a;
20 wire       b;
21 wire       ci;
22 wire       sum;
23 wire       co;
24
25 endmodule
```

You could download file comment.v [here](#)

### Case Sensitivity

Verilog HDL is case sensitive

- Lower case letters are unique from upper case letters
- All Verilog keywords are lower case

### Examples of Unique names

```
1 input      // a Verilog Keyword
2 wire       // a Verilog Keyword
3 WIRE       // a unique name ( not a keyword)
4 Wire       // a unique name (not a keyword)
```

You could download file unique\_names.v [here](#)

**NOTE :** Never use Verilog keywords as unique names, even if the case is different.

### ❖ Identifiers

Identifiers are names used to give an object, such as a register or a function or a module, a name so that it can be referenced from other places in a description.

- Identifiers must begin with an alphabetic character or the underscore character (**a-z A-Z \_**)
- Identifiers may contain alphabetic characters, numeric characters, the underscore, and the dollar sign (**a-z A-Z 0-9 \_ \$**)
- Identifiers can be up to 1024 characters long.

### ✦ Examples of legal identifiers

```
data_input mu
clk_input my$clk
i386 A
```

### ❖ Escaped Identifiers

Verilog HDL allows any character to be used in an identifier by escaping the identifier. Escaped identifiers provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

- Escaped identifiers begin with the back slash ( \ )
- Entire identifier is escaped by the back slash.
- Escaped identifier is terminated by white space (Characters such as commas, parentheses, and semicolons become part of the escaped identifier unless preceded by a white space)
- Terminate escaped identifiers with white space, otherwise characters that should follow the identifier are considered as part of it.

### ✦ Examples of escape identifiers

Verilog does not allow to identifier to start with a numeric character. So if you really want to use a identifier to start with a numeric value then use a escape character as shown below.

```
1 // There must be white space after the
```

```

2 // string which uses escape character
3 module \ldff (
4   q,          // Q output
5   \q~,       // Q_out output
6   d,          // D input
7   cl$k,       // CLOCK input
8   \reset*    // Reset input
9 );
10
11 input d, cl$k, \reset* ;
12 output q, \q~ ;
13
14 endmodule

```

You could download file escape\_id.v [here](#)

## Numbers in Verilog

You can specify constant numbers in decimal, hexadecimal, octal, or binary format. Negative numbers are represented in 2's complement form. When used in a number, the question mark (?) character is the Verilog alternative for the z character. The underscore character (\_) is legal anywhere in a number except as the first character, where it is ignored.

## Integer Numbers

Verilog HDL allows integer numbers to be specified as

- Sized or unsized numbers (Unsize size is 32 bits)
- In a radix of binary, octal, decimal, or hexadecimal
- Radix and hex digits (a,b,c,d,e,f) are case insensitive
- Spaces are allowed between the size, radix and value

Syntax: <size>'<radix><value>;

## Example of Integer Numbers

| Integer    | Stored as                        |
|------------|----------------------------------|
| 1          | 00000000000000000000000000000001 |
| 8'hAA      | 10101010                         |
| 6'b10_0011 | 100011                           |
| 'hF        | 00000000000000000000000000001111 |

Verilog expands <value> filling the specified <size> by working from right-to-left

- When <size> is smaller than <value>, then leftmost bits of <value> are truncated
- When <size> is larger than <value>, then leftmost bits are filled, based on the value of the leftmost bit in <value>.
  - Leftmost '0' or '1' are filled with '0'
  - Leftmost 'Z' are filled with 'Z'
  - Leftmost 'X' are filled with 'X'

**Note :** X Stands for unknown and Z stands for high impedance, 1 for logic high or 1 and 0 for logic low or 0.

#### ◆ Example of Integer Numbers

| Integer | Stored as               |
|---------|-------------------------|
| 6'hCA   | 001010                  |
| 6'hA    | 001010                  |
| 16'bZ   | <i>ZZZZZZZZZZZZZZZZ</i> |
| 8'bx    | xxxxxxxx                |

#### ◆ Real Numbers

- Verilog supports real constants and variables
- Verilog converts real numbers to integers by rounding
- Real Numbers can not contain 'Z' and 'X'
- Real numbers may be specified in either decimal or scientific notation
- < value >.< value >
- < mantissa >E< exponent >
- Real numbers are rounded off to the nearest integer when assigning to an integer.

#### ◆ Example of Real Numbers

| Real Number | Decimal notation |
|-------------|------------------|
| 1.2         | 1.2              |
| 0.6         | 0.6              |
| 3.5E6       | 3,500000.0       |



### Signed and Unsigned Numbers

Verilog Supports both types of numbers, but with certain restrictions. Like in C language we don't have int and uint types to say if a number is signed integer or unsigned integer.

Any number that does not have negative sign prefix is a positive number. Or indirect way would be "Unsigned".

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus they become signed numbers. Verilog internally represents negative numbers in 2's complement format. An optional signed specifier can be added for signed arithmetic.

### Examples

| Number        | Description                        |
|---------------|------------------------------------|
| 32'hDEAD_BEEF | Unsigned or signed positive number |
| -14'h1234     | Signed negative number             |

The example file below shows how Verilog treats signed and unsigned numbers.

```

1 module signed_number;
2
3 reg [31:0] a;
4
5 initial begin
6   a = 14'h1234;
7   $display ("Current Value of a = %h", a);
8   a = -14'h1234;
9   $display ("Current Value of a = %h", a);
10  a = 32'hDEAD_BEEF;
11  $display ("Current Value of a = %h", a);
12  a = -32'hDEAD_BEEF;
13  $display ("Current Value of a = %h", a);
14  #10 $finish;
15 end
16
17 endmodule
  
```

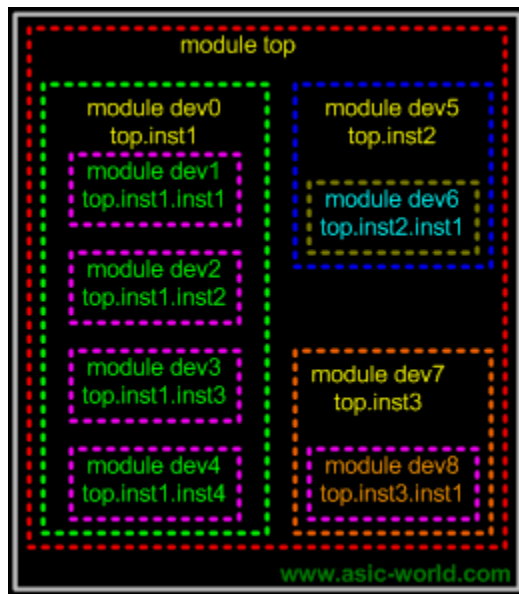
You could download file signed\_number.v [here](#)

Current Value of a = 00001234

Current Value of a = ffffedcc  
 Current Value of a = deadbeef  
 Current Value of a = 21524111

## Modules

- Modules are the building blocks of Verilog designs
- You create the design hierarchy by instantiating modules in other modules.
- You instance a module when you use that module in another, higher-level module.



## Ports

- Ports allow communication between a module and its environment.
- All but the top-level modules in a hierarchy have ports.
- Ports can be associated by order or by name.

You declare ports to be input, output or inout. The port declaration syntax is :

```
input [range_val:range_var] list_of_identifiers;
output [range_val:range_var] list_of_identifiers;
```

inout [range\_val:range\_var] list\_of\_identifiers;

**NOTE :** As a good coding practice, there should be only one port identifier per line, as shown below

### ✦ Examples : Port Declaration

```

1 input          clk          ; // clock input
2 input  [15:0]   data_in     ; // 16 bit data input bus
3 output [7:0]    count       ; // 8 bit counter output
4 inout         data_bi      ; // Bi-Directional data bus

```

You could download file port\_declare.v [here](#)

### ✦ Examples : A complete Example in Verilog

```

1 module addbit (
2   a      , // first input
3   b      , // Second input
4   ci     , // Carry input
5   sum    , // sum output
6   co     , // carry output
7 );
8 //Input declaration
9 input a;
10 input b;
11 input ci;
12 //Output declaration
13 output sum;
14 output co;
15 //Port Data types
16 wire a;
17 wire b;
18 wire ci;
19 wire sum;
20 wire co;
21 //Code starts here
22 assign {co,sum} = a + b + ci;
23
24 endmodule // End of Module addbit

```

You could download file addbit.v [here](#)

### ✦ Modules connected by port order (implicit)

Here order should match correctly. Normally it's not a good idea to connect ports implicitly. It could cause problem in debug (for example:

locating the port which is causing a compile error), when any port is added or deleted.

```

1 //-----
2 // This is simple adder Program
3 // Design Name : adder_implicit
4 // File Name : adder_implicit.v
5 // Function : This program shows how implicit
6 //           port connection are done
7 // Coder : Deepak Kumar Tala
8 //-----
9 module adder_implicit (
10 result      , // Output of the adder
11 carry       , // Carry output of adder
12 r1          , // first input
13 r2          , // second input
14 ci          , // carry input
15 );
16
17 // Input Port Declarations
18 input  [3:0]  r1      ;
19 input  [3:0]  r2      ;
20 input                ci      ;
21
22 // Output Port Declarations
23 output [3:0]  result   ;
24 output                carry ;
25
26 // Port Wires
27 wire  [3:0]  r1      ;
28 wire  [3:0]  r2      ;
29 wire                ci      ;
30 wire  [3:0]  result   ;
31 wire                carry ;
32
33 // Internal variables
34 wire                c1      ;
35 wire                c2      ;
36 wire                c3      ;
37
38 // Code Starts Here
39 addbit u0 (
40 r1[0]      ,
41 r2[0]      ,
42 ci         ,
43 result[0]  ,
44 c1         ,
45 );
  
```

```
46
47 addbit u1 (
48 r1[1]      ,
49 r2[1]      ,
50 c1         ,
51 result[1]  ,
52 c2
53 );
54
55 addbit u2 (
56 r1[2]      ,
57 r2[2]      ,
58 c2         ,
59 result[2]  ,
60 c3
61 );
62
63 addbit u3 (
64 r1[3]      ,
65 r2[3]      ,
66 c3         ,
67 result[3]  ,
68 carry
69 );
70
71 endmodule // End Of Module adder
```

You could download file adder\_implicit.v [here](#)

#### ✦ Modules connected by name

Here the name should match with the leaf module, the order is not important.

```
1 //-----
2 // This is simple adder Program
3 // Design Name : adder_explicit
4 // File Name  : adder_explicit.v
5 // Function   : Here the name should match
6 // with the leaf module, the order is not important.
7 // Coder      : Deepak Kumar Tala
8 //-----
9 module adder_explicit (
10 result      , // Output of the adder
11 carry       , // Carry output of adder
12 r1          , // first input
13 r2          , // second input
14 ci          , // carry input
15 );
```

```

16
17 // Input Port Declarations
18 input    [3:0]    r1        ;
19 input    [3:0]    r2        ;
20 input                    ci        ;
21
22 // Output Port Declarations
23 output    [3:0]    result    ;
24 output                    carry    ;
25
26 // Port Wires
27 wire      [3:0]    r1        ;
28 wire      [3:0]    r2        ;
29 wire                        ci        ;
30 wire      [3:0]    result    ;
31 wire                        carry    ;
32
33 // Internal variables
34 wire                        c1        ;
35 wire                        c2        ;
36 wire                        c3        ;
37
38 // Code Starts Here
39 addbit u0 (
40 .a            (r1[0])        ,
41 .b            (r2[0])        ,
42 .ci           (ci)           ,
43 .sum          (result[0])    ,
44 .co           (c1)           ,
45 );
46
47 addbit u1 (
48 .a            (r1[1])        ,
49 .b            (r2[1])        ,
50 .ci           (c1)           ,
51 .sum          (result[1])    ,
52 .co           (c2)           ,
53 );
54
55 addbit u2 (
56 .a            (r1[2])        ,
57 .b            (r2[2])        ,
58 .ci           (c2)           ,
59 .sum          (result[2])    ,
60 .co           (c3)           ,
61 );
62
63 addbit u3 (
64 .a            (r1[3])        ,
65 .b            (r2[3])        ,

```

```
66 .ci          (c3)          ,
67 .sum         (result[3])   ,
68 .co          (carry)
69 );
70
71 endmodule // End Of Module adder
```

You could download file adder\_explicit.v [here](#)

### ✦ Instantiating a module

```
1 //-----
2 // This is simple parity Program
3 // Design Name : parity
4 // File Name   : parity.v
5 // Function    : This program shows how a verilog
6 //               primitive/module port connection are done
7 // Coder       : Deepak
8 //-----
9 module parity (
10 a      , // First input
11 b      , // Second input
12 c      , // Third Input
13 d      , // Fourth Input
14 y      // Parity output
15 );
16
17 // Input Declaration
18 input  a      ;
19 input  b      ;
20 input  c      ;
21 input  d      ;
22 // Output Declaration
23 output y      ;
24 // port data types
25 wire  a      ;
26 wire  b      ;
27 wire  c      ;
28 wire  d      ;
29 wire  y      ;
30 // Internal variables
31 wire  out_0   ;
32 wire  out_1   ;
33
34 // Code starts Here
35 xor u0 (out_0,a,b);
36
37 xor u1 (out_1,c,d);
```

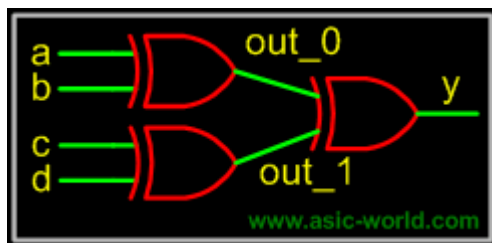
```

38
39 xor u2 (y,out_0,out_1);
40
41 endmodule // End Of Module parity
  
```

You could download file parity.v [here](#)

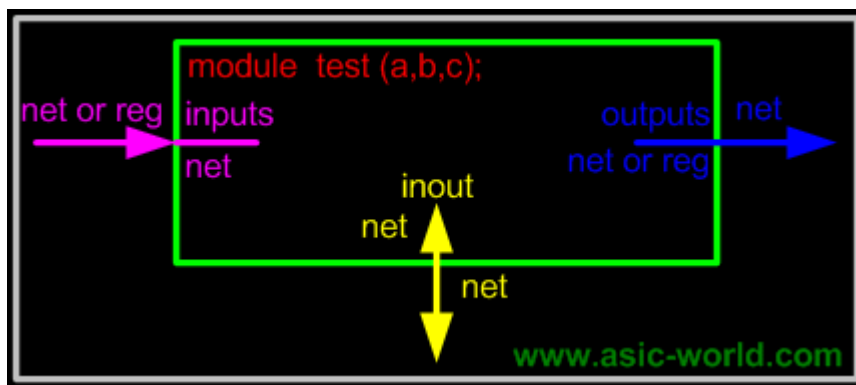
**Question :** What is the difference between u0 in module adder and u0 in module parity?

### ✦ Schematic



### ● Port Connection Rules

- Inputs : internally must always be of type net, externally the inputs can be connected to a variable of type reg or net.
- Outputs : internally can be of type net or reg, externally the outputs must be connected to a variable of type net.
- Inouts : internally or externally must always be type net, can only be connected to a variable net type.



- Width matching : It is legal to connect internal and external ports of different sizes. But beware, synthesis tools could report problems.



- Unconnected ports : unconnected ports are allowed by using a ",".
- The net data types are used to connect structure.
- A net data type is required if a signal can be driven a structural connection.

#### Example - Implicit Unconnected Port

```
1 module implicit();
2 reg clk,d,rst,pre;
3 wire q;
4
5 // Here second port is not connected
6 dff u0 ( q,,clk,d,rst,pre);
7
8 endmodule
9
10 // D fli-flop
11 module dff (q, q_bar, clk, d, rst, pre);
12 input clk, d, rst, pre;
13 output q, q_bar;
14 reg q;
15
16 assign q_bar = ~q;
17
18 always @ (posedge clk)
19 if (rst == 1'b1) begin
20     q <= 0;
21 end else if (pre == 1'b1) begin
22     q <= 1;
23 end else begin
24     q <= d;
25 end
26
27 endmodule
```

You could download file implicit.v [here](#)

#### Example - Explicit Unconnected Port

```
1 module explicit();
2 reg clk,d,rst,pre;
3 wire q;
4
5 // Here q_bar is not connected
```

```

6 // We can connect ports in any order
7 dff u0 (
8   .q      (q) ,
9   .d      (d) ,
10  .clk     (clk) ,
11  .q_bar   (),
12  .rst     (rst) ,
13  .pre     (pre)
14 );
15
16 endmodule
17
18 // D fli-flop
19 module dff (q, q_bar, clk, d, rst, pre);
20 input clk, d, rst, pre;
21 output q, q_bar;
22 reg q;
23
24 assign q_bar = ~q;
25
26 always @ (posedge clk)
27 if (rst == 1'b1) begin
28   q <= 0;
29 end else if (pre == 1'b1) begin
30   q <= 1;
31 end else begin
32   q <= d;
33 end
34
35 endmodule
  
```

You could download file explicit.v [here](#)

## Hierarchical Identifiers

Hierarchical path names are based on the top module identifier followed by module instant identifiers, separated by periods.

This is useful basically when we want to see the signal inside a lower module, or want to force a value inside an internal module. The example below shows how to monitor the value of an internal module signal.



### Example

```

1 //-----
2 // This is simple adder Program
3 // Design Name : adder_hier
  
```

```

4 // File Name : adder_hier.v
5 // Function : This program shows verilog hier path works
6 // Coder : Deepak
7 //-----
8 `include "addbit.v"
9 module adder_hier (
10 result , // Output of the adder
11 carry , // Carry output of adder
12 r1 , // first input
13 r2 , // second input
14 ci // carry input
15 );
16
17 // Input Port Declarations
18 input [3:0] r1 ;
19 input [3:0] r2 ;
20 input ci ;
21
22 // Output Port Declarations
23 output [3:0] result ;
24 output carry ;
25
26 // Port Wires
27 wire [3:0] r1 ;
28 wire [3:0] r2 ;
29 wire ci ;
30 wire [3:0] result ;
31 wire carry ;
32
33 // Internal variables
34 wire c1 ;
35 wire c2 ;
36 wire c3 ;
37
38 // Code Starts Here
39 addbit u0 (r1[0],r2[0],ci,result[0],c1);
40 addbit u1 (r1[1],r2[1],c1,result[1],c2);
41 addbit u2 (r1[2],r2[2],c2,result[2],c3);
42 addbit u3 (r1[3],r2[3],c3,result[3],carry);
43
44 endmodule // End Of Module adder
45
46 module tb();
47
48 reg [3:0] r1,r2;
49 reg ci;
50 wire [3:0] result;
51 wire carry;
52

```

```

53 // Drive the inputs
54 initial begin
55     r1 = 0;
56     r2 = 0;
57     ci = 0;
58     #10 r1 = 10;
59     #10 r2 = 2;
60     #10 ci = 1;
61     #10 $display("+-----+");
62     $finish;
63 end
64
65 // Connect the lower module
66 adder_hier U (result,carry,r1,r2,ci);
67
68 // Hier demo here
69 initial begin
70     $display("+-----+");
71     $display("| r1 | r2 | ci | u0.sum | u1.sum | u2.sum | u3.sum |");
72     $display("+-----+");
73     $monitor("| %h | %h | %h | %h | %h | %h | %h |");
74     r1,r2,ci, tb.U.u0.sum, tb.U.u1.sum, tb.U.u2.sum,
    tb.U.u3.sum);
75 end
76
77 endmodule
  
```

You could download file adder\_hier.v [here](#)

```

+-----+
| r1 | r2 | ci | u0.sum | u1.sum | u2.sum | u3.sum |
+-----+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | 0 | 0 | 0 | 1 | 0 | 1 |
| a | 2 | 0 | 0 | 0 | 1 | 1 |
| a | 2 | 1 | 1 | 0 | 1 | 1 |
+-----+
  
```



## Data Types

Verilog Language has two primary data types:

- **Nets** - represent structural connections between components.
- **Registers** - represent variables used to store data.

Every signal has a data type associated with it:

- **Explicitly declared** with a declaration in your Verilog code.
- **Implicitly declared** with no declaration when used to connect structural building blocks in your code. Implicit declaration is always a net type "wire" and is one bit wide.



### Types of Nets

Each net type has a functionality that is used to model different types of hardware (such as PMOS, NMOS, CMOS, etc)

| Net Data Type    | Functionality   |
|------------------|---|
| wire, tri        | Interconnecting wire - no special resolution function   |
| wor, trior       | Wired outputs OR together (models ECL)                  |
| wand, triand     | Wired outputs AND together (models open-collector)      |
| tri0, tri1       | Net pulls-down or pulls-up when not driven              |
| supply0, supply1 | Net has a constant logic 0 or logic 1 (supply strength) |
| trireg           | Retains last value, when driven by z (tristate).        |

**Note :** Of all net types, wire is the one which is most widely used.



### Example - wor

```

1 module test_wor();
2
3   wor a;
4   reg b, c;
5
6   assign a = b;
7   assign a = c;
8
9   initial begin
10    $monitor ("%g a = %b b = %b c = %b", $time, a, b, c);
11    #1 b = 0;
12    #1 c = 0;
13    #1 b = 1;
  
```

```
14    #1 b = 0;  
15    #1 c = 1;  
16    #1 b = 1;  
17    #1 b = 0;  
18    #1 $finish;  
19 end  
20  
21 endmodule
```

You could download file test\_wor.v [here](#)

### Simulator Output

```
0 a=x b=x c=x  
1 a=x b=0 c=x  
2 a=0 b=0 c=0  
3 a=1 b=1 c=0  
4 a=0 b=0 c=0  
5 a=1 b=0 c=1  
6 a=1 b=1 c=1  
7 a=1 b=0 c=1
```



### Example - wand

```
1 module test_wand();  
2  
3 wand a;  
4 reg b, c;  
5  
6 assign a = b;  
7 assign a = c;  
8  
9 initial begin  
10    $monitor ("%g a = %b b = %b c = %b", $time, a, b, c);  
11    #1 b = 0;  
12    #1 c = 0;  
13    #1 b = 1;  
14    #1 b = 0;  
15    #1 c = 1;  
16    #1 b = 1;  
17    #1 b = 0;  
18    #1 $finish;  
19 end  
20  
21 endmodule
```

You could download file test\_wand.v [here](#)

**Simulator Output**

```
0 a=x b=x c=x
1 a=0 b=0 c=x
2 a=0 b=0 c=0
3 a=0 b=1 c=0
4 a=0 b=0 c=0
5 a=0 b=0 c=1
6 a=1 b=1 c=1
7 a=0 b=0 c=1
```

✦ **Example - tri**

```
1 module test_tri();
2
3 tri a;
4 reg b, c;
5
6 assign a = (b) ? c : 1'bz;
7
8 initial begin
9     $monitor ("%g a = %b b = %b c = %b", $time, a, b, c);
10    b = 0;
11    c = 0;
12    #1 b = 1;
13    #1 b = 0;
14    #1 c = 1;
15    #1 b = 1;
16    #1 b = 0;
17    #1 $finish;
18 end
19
20 endmodule
```

You could download file test\_tri.v [here](#)

**Simulator Output**

```
0 a=z b=0 c=0
1 a=0 b=1 c=0
2 a=z b=0 c=0
3 a=z b=0 c=1
4 a=1 b=1 c=1
5 a=z b=0 c=1
```

✦ **Example - trireg**

```

1 module test_trireg();
2
3   trireg a;
4   reg b, c;
5
6   assign a = (b) ? c : 1'bz;
7
8   initial begin
9     $monitor ("%g a = %b b = %b c = %b", $time, a, b, c);
10    b = 0;
11    c = 0;
12    #1 b = 1;
13    #1 b = 0;
14    #1 c = 1;
15    #1 b = 1;
16    #1 b = 0;
17    #1 $finish;
18  end
19
20 endmodule

```

You could download file test\_trireg.v [here](#)

### Simulator Output

```

0 a = x b = 0 c = 0
1 a = 0 b = 1 c = 0
2 a = 0 b = 0 c = 0
3 a = 0 b = 0 c = 1
4 a = 1 b = 1 c = 1
5 a = 1 b = 0 c = 1

```



### Register Data Types

- Registers store the last value assigned to them until another assignment statement changes their value.
- Registers represent data storage constructs.
- You can create regs arrays called memories.
- register data types are used as variables in procedural blocks.
- A register data type is required if a signal is assigned a value within a procedural block
- Procedural blocks begin with keyword initial and always.

| Data Types | Functionality     |
|------------|-------------------|
| reg        | Unsigned variable |



|         |  |
|---------|--|
| integer | Signed variable - 32 bits                |
| time    | Unsigned integer - 64 bits               |
| real    | Double precision floating point variable |

**Note :** Of all register types, reg is the one which is most widely used

### Strings

A string is a sequence of characters enclosed by double quotes and all contained on a single line. Strings used as operands in expressions and assignments are treated as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character. To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character. Strings can be manipulated using the standard operators.

When a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values.

Certain characters can be used in strings only when preceded by an introductory character called an escape character. The following table lists these characters in the right-hand column together with the escape sequence that represents the character in the left-hand column.



### Special Characters in Strings

| Character | Description   |
|-----------|---|
| \n        | New line character  |
| \t        | Tab character   |
| \\        | Backslash (\) character   |
| \"        | Double quote (") character                                      |
| \ddd      | A character specified in 1-3 octal digits ( $0 \leq d \leq 7$ ) |

|    |                       |
|----|-----------------------|
| %% | Percent (%) character |
|----|-----------------------|

### ✦ Example

```

1 //-----
2 // Design Name : strings
3 // File Name  : strings.v
4 // Function   : This program shows how string
5 //             can be stored in reg
6 // Coder💎    : Deepak Kumar Tala
7 //-----
8 module strings();
9 // Declare a register variable that is 21 bytes
10 reg [8*21:0] string ;
11
12 initial begin
13   string = "This is sample string";
14   $display ("%s \n", string);
15 end
16
17 endmodule
  
```

You could download file strings.v [here](#)

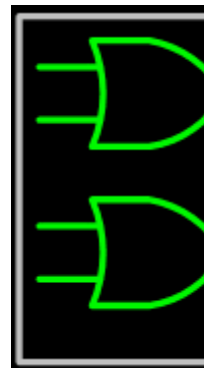
|                       |
|-----------------------|
| This is sample string |
|-----------------------|

## Introduction

Verilog has built-in primitives and gates. These are used in post-synthesis. These cells are called as SDF cells. They are used in synthesis tool and also in Verilog.

Note : RTL uses built-in library cells in Verilog.

## Gate Primitives



The gates have  
1st terminal in  
terminals are

| Gate |
|------|
| and  |
| nand |
| or   |
| nor  |
| xor  |
| xnor |



## Examples

```

1 module g
2
3 wire out0
4 wire out1
5 wire out2
6 reg in1,
7
8 not U1 (ou
9 and U2 (o
10 xor U3 (ou
11
12 initial beg
13 $monito
14 "in1=%l
15 in1,i
16 in1 = 0
17 in2 = 0
  
```

```

18   in3 = 0
19   in4 = 0
20   #1   in1
21   #1   in2
22   #1   in3
23   #1   in4
24   #1   $fin
25 end
26
27 endmodule

```

You could download

```

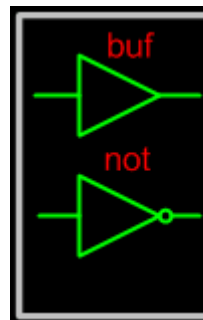
in1 = 0 in2 = 0 in
in1 = 1 in2 = 0 in
in1 = 1 in2 = 1 in
in1 = 1 in2 = 1 in
in1 = 1 in2 = 1 in

```

## Transmission

Transmission  
resistive.

Syntax: keyw



## Gate

not

buf

bufif0

bufif1

notif0

notif1

Transmission  
 have a contro  
 separate drive



## Examples

```

1 module t
2
3 reg data_
4 wire data_
5
6 bufif0 U1
7 buf U2 (
8 not U3 (ou
9
10 initial beg
11 $monito
12 "@%
data_bus=%b",
13 $time
data_bus);
14 data_e
15 in = 0;
16 #4 dat
17 #8 $fin
18 end
19
20 always #2
21
22 endmodul
  
```

You could downl

```

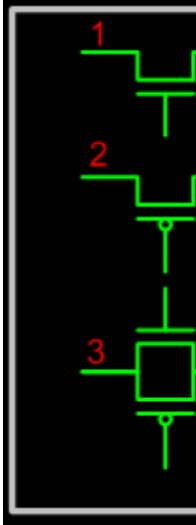
@0 in = 0 data_
@2 in = 1 data_
@4 in = 0 data_
@6 in = 1 data_
@8 in = 0 data_
@10 in = 1 data_
  
```



## Switch Primi

There are six  
 in Verilog, nm  
 resistive vers  
 switches have

**Syntax:** keyword



### Gate

1. pmos

1. rpms

2. nmos

2. rnmos

3. cmos

3. rcmos

4. tranif1

4. tranif0

5. rtranif1

5. rtranif0

6. tran

6. rtran

7. pullup

8. pulldown

Transmission  
resistive. Res  
appears on th  
signals from s  
result in high



## Examples

```

1 module sw
2
3 wire net1
4 wire net4
5
6 tranif0 my_
7 rtranif1 my_
8
9 endmodule
  
```

You could downl

Transmission  
 have a contro  
 separate driv  
 Resistive dev  
 the output by  
 source to dra  
 impedance ou



## Logic Values

The Verilog H

### Logic Value

0

1

z or Z

x or X



## Verilog Streng

### Strength Lev

7 Supply Drive

6 Strong Pull

5 Pull Drive

4 Large Capac

3 Weak Drive

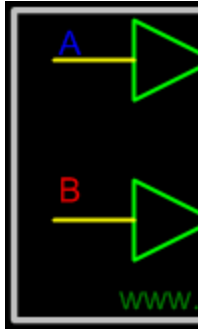
2 Medium Cap

1 Small Capac

0 Hi Impedanc



**Example : St**



Two buffers th

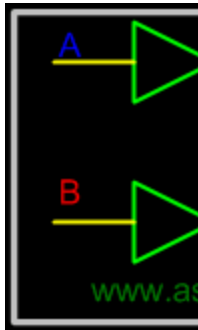
A : Pull 1

**B : Supply 0**

Since supply



**Example 2 : S**



Two buffers th

**A : Supply 1**

B : Large 1

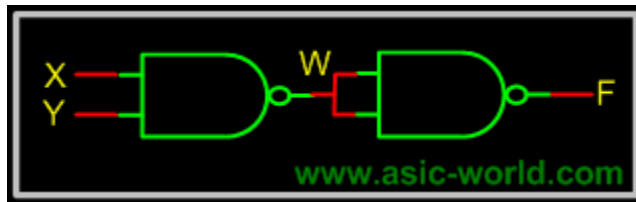
Since Supply  
of A

## Designing Using Primitives

Designing using primitives is used only in library development, where the ASIC vendor provides the ASIC library Verilog description, using Verilog primitives and user defined primitive



### AND Gate from NAND Gate



#### Code

```

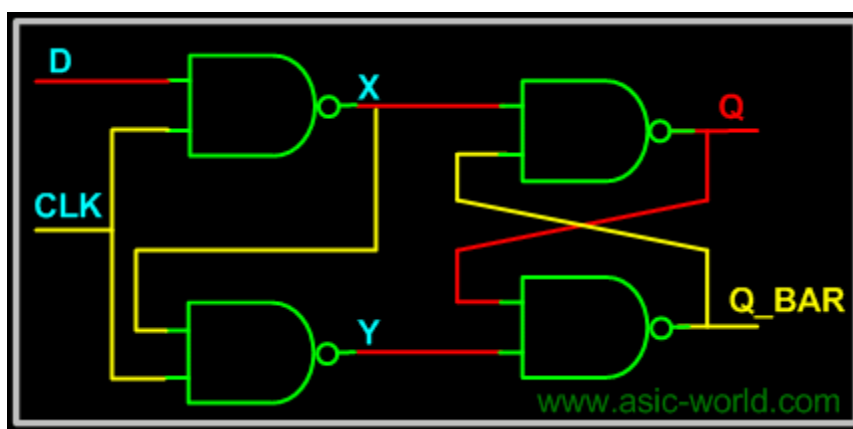
1 // Structural model of AND gate from two NANDS
2 module and_from_nand();
3
4 reg X, Y;
5 wire F, W;
6 // Two instantiations of the module NAND
7 nand U1(W,X, Y);
8 nand U2(F, W, W);
9
10 // Testbench Code
11 initial begin
12     $monitor ("X = %b Y = %b F = %b", X, Y, F);
13     X = 0;
14     Y = 0;
15     #1 X = 1;
16     #1 Y = 1;
17     #1 X = 0;
18     #1 $finish;
19 end
20
21 endmodule
  
```

You could download file and\_from\_nand.v [here](#)

```

X = 0 Y = 0 F = 0
X = 1 Y = 0 F = 0
X = 1 Y = 1 F = 1
X = 0 Y = 1 F = 0
  
```

### D-Flip flop from NAND Gate



### ✦ Verilog Code

```

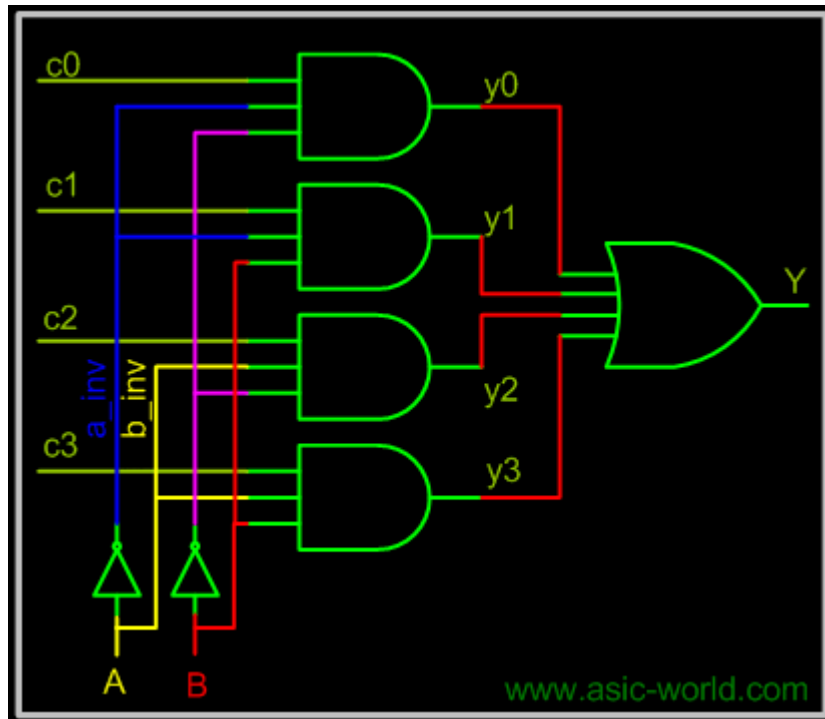
1  module dff_from_nand();
2  wire Q,Q_BAR;
3  reg D,CLK;
4
5  nand U1 (X,D,CLK) ;
6  nand U2 (Y,X,CLK) ;
7  nand U3 (Q,Q_BAR,X) ;
8  nand U4 (Q_BAR,Q,Y) ;
9
10 // Testbench of above code
11 initial begin
12     $monitor ("CLK = %b D = %b Q = %b Q_BAR = %b",CLK, D, Q, Q_BAR) ;
13     CLK = 0;
14     D = 0;
15     #3 D = 1;
16     #3 D = 0;
17     #3 $finish;
18 end
19
20 always #2 CLK = ~CLK;
21
22 endmodule
  
```

You could download file dff\_from\_nand.v [here](#)

```

CLK = 0 D = 0 Q = x Q_BAR = x
CLK = 1 D = 0 Q = 0 Q_BAR = 1
CLK = 1 D = 1 Q = 1 Q_BAR = 0
CLK = 0 D = 1 Q = 1 Q_BAR = 0
CLK = 1 D = 0 Q = 0 Q_BAR = 1
CLK = 0 D = 0 Q = 0 Q_BAR = 1
  
```

### ◆ Multiplexer from primitives



### ◆ Verilog Code

```

1 module mux_from_gates ();
2 reg c0,c1,c2,c3,A,B;
3 wire Y;
4 //Invert the sel signals
5 not (a_inv, A);
6 not (b_inv, B);
7 // 3-input AND gate
8 and (y0,c0,a_inv,b_inv);
9 and (y1,c1,a_inv,B);
10 and (y2,c2,A,b_inv);
11 and (y3,c3,A,B);
12 // 4-input OR gate
13 or (Y, y0,y1,y2,y3);
14
15 // Testbench Code goes here
16 initial begin
17     $monitor (
18         "c0 = %b c1 = %b c2 = %b c3 = %b A = %b B = %b Y = %b",

```

```

19    c0, c1, c2, c3, A, B, Y);
20    c0 = 0;
21    c1 = 0;
22    c2 = 0;
23    c3 = 0;
24    A = 0;
25    B = 0;
26    #1 A = 1;
27    #2 B = 1;
28    #4 A = 0;
29    #8 $finish;
30 end
31
32 always #1 c0 = ~c0;
33 always #2 c1 = ~c1;
34 always #3 c2 = ~c2;
35 always #4 c3 = ~c3;
36
37 endmodule
  
```

You could download file mux\_from\_gates.v [here](#)

```

c0 = 0 c1 = 0 c2 = 0 c3 = 0 A = 0 B = 0 Y = 0
c0 = 1 c1 = 0 c2 = 0 c3 = 0 A = 1 B = 0 Y = 0
c0 = 0 c1 = 1 c2 = 0 c3 = 0 A = 1 B = 0 Y = 0
c0 = 1 c1 = 1 c2 = 1 c3 = 0 A = 1 B = 1 Y = 0
c0 = 0 c1 = 0 c2 = 1 c3 = 1 A = 1 B = 1 Y = 1
c0 = 1 c1 = 0 c2 = 1 c3 = 1 A = 1 B = 1 Y = 1
c0 = 0 c1 = 1 c2 = 0 c3 = 1 A = 1 B = 1 Y = 1
c0 = 1 c1 = 1 c2 = 0 c3 = 1 A = 0 B = 1 Y = 1
c0 = 0 c1 = 0 c2 = 0 c3 = 0 A = 0 B = 1 Y = 0
c0 = 1 c1 = 0 c2 = 1 c3 = 0 A = 0 B = 1 Y = 0
c0 = 0 c1 = 1 c2 = 1 c3 = 0 A = 0 B = 1 Y = 1
c0 = 1 c1 = 1 c2 = 1 c3 = 0 A = 0 B = 1 Y = 1
c0 = 0 c1 = 0 c2 = 0 c3 = 1 A = 0 B = 1 Y = 0
c0 = 1 c1 = 0 c2 = 0 c3 = 1 A = 0 B = 1 Y = 0
c0 = 0 c1 = 1 c2 = 0 c3 = 1 A = 0 B = 1 Y = 1
  
```

### Gate and Switch delays

In real circuits, logic gates have delays associated with them. Verilog provides the means to associate delays with gates.

- Rise, Fall and Turn-off delays.
- Minimal, Typical, and Maximum delays.

In Verilog delays can be introduced with `#'num'` as in the examples below, where `#` is character to introduce delay, and `'num'` is the number of ticks simulator should delay statement execution.

- `#1 a = b` : Delay by 1, i.e. execute after 1 tick
- `#2 not (a,b)` : Delay by 2 all assignments made to a.

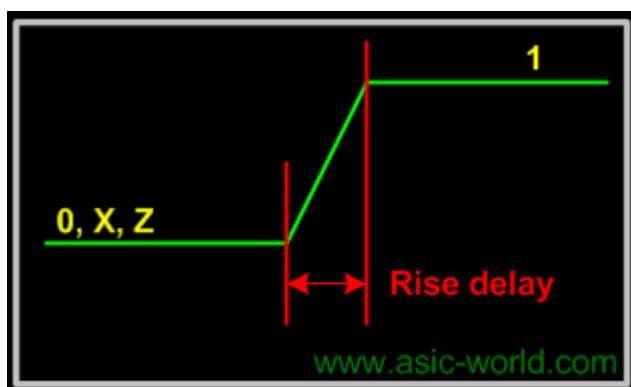
Real transistors have resolution delays between the input and output. This is modeled in Verilog by specifying one or more delays for the rise, fall, turn-on and turn off time separately using commas.

**Syntax:** keyword `#(delay{s})` unique\_name (node specifications);

| Switch element         | Number Delays | Of Specified delays                           |
|------------------------|---------------|---|
| Switch                 | 1             | Rise, fall and turn-off times of equal length |
|                        | 2             | Rise and fall times                           |
|                        | 3             | Rise, fall and turn off                       |
| (r)tranif0, (r)tranif1 | 1             | both turn on and turn off                     |
|                        | 2             | turn on, turn off                             |
| (r)tran                | 0             | None allowed                                  |

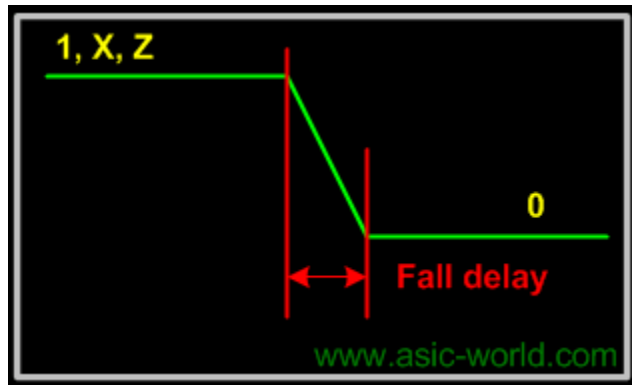
### ❖ Rise Delay

The rise delay is associated with a gate output transition to 1 from another value (0, x, z).



### ❖ Fall Delay

The fall delay is associated with a gate output transition to 0 from another value (1, x, z).



#### ❖ Turn-off Delay

The Turn-off delay is associated with a gate output transition to z from another value.

#### ❖ Min Value

The min value is the minimum delay value that the gate is expected to have.

#### ❖ Typ Value

The typ value is the typical delay value that the gate is expected to have.

#### ❖ Max Value

The max value is the maximum delay value that the gate is expected to have.

#### ❖ Example

Below are some examples to show the usage of delays.

#### ✦ Example - Single Delay

```

1 module buf_gate ();
2 reg in;
3 wire out;
4
5 buf #(5) (out,in);
6
7 initial begin
8   $monitor ("Time = %g in = %b out=%b", $time, in, out);
9   in = 0;
10  #10 in = 1;
11  #10 in = 0;
12  #10 $finish;

```

```

13 end
14
15 endmodule

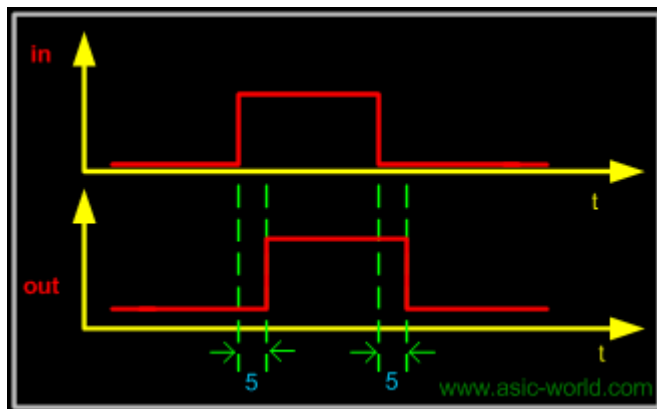
```

You could download file buf\_gate.v [here](#)

```

Time = 0 in = 0 out=x
Time = 5 in = 0 out=0
Time = 10 in = 1 out=0
Time = 15 in = 1 out=1
Time = 20 in = 0 out=1
Time = 25 in = 0 out=0

```



### ✦ Example - Two Delays

```

1 module buf_gate1 ();
2   reg in;
3   wire out;
4
5   buf #(2,3) (out,in);
6
7   initial begin
8     $monitor ("Time = %g in = %b out=%b", $time, in, out);
9     in = 0;
10    #10 in = 1;
11    #10 in = 0;
12    #10 $finish;
13  end
14
15 endmodule

```

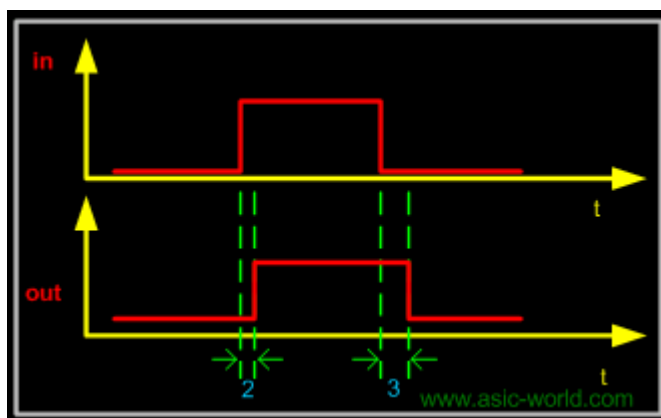
You could download file buf\_gate1.v [here](#)

```

Time = 0 in = 0 out=x
Time = 3 in = 0 out=0

```

Time = 10 in = 1 out=0  
 Time = 12 in = 1 out=1  
 Time = 20 in = 0 out=1  
 Time = 23 in = 0 out=0



### ◆ Example - All Delays

```

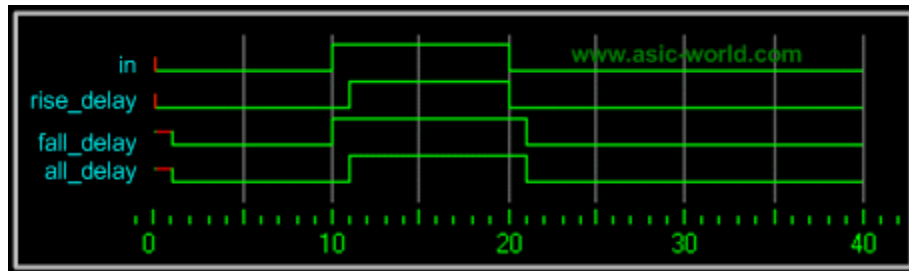
1 module delay();
2   reg in;
3   wire rise_delay, fall_delay, all_delay;
4
5   initial begin
6     $monitor (
7       "Time=%g in=%b rise_delay=%b fall_delay=%b all_delay=%b",
8       $time, in, rise_delay, fall_delay, all_delay);
9     in = 0;
10    #10 in = 1;
11    #10 in = 0;
12    #20 $finish;
13  end
14
15  buf #(1,0) U_rise (rise_delay,in);
16  buf #(0,1) U_fall (fall_delay,in);
17  buf #1 U_all (all_delay,in);
18
19 endmodule
  
```

You could download file delay.v [here](#)

Time = 0 in = 0 rise\_delay = 0 fall\_delay = x all\_delay = x  
 Time = 1 in = 0 rise\_delay = 0 fall\_delay = 0 all\_delay = 0  
 Time = 10 in = 1 rise\_delay = 0 fall\_delay = 1 all\_delay = 0  
 Time = 11 in = 1 rise\_delay = 1 fall\_delay = 1 all\_delay = 1  
 Time = 20 in = 0 rise\_delay = 0 fall\_delay = 1 all\_delay = 1



Time = 21 in = 0 rise\_delay = 0 fall\_delay = 0 all\_delay = 0



### ✦ Example - Complex Example

```

1 module delay_example();
2
3 wire out1,out2,out3,out4,out5,out6;
4 reg b,c;
5
6 // Delay for all transitions
7 or #5 u_or (out1,b,c);
8 // Rise and fall delay
9 and #(1,2) u_and (out2,b,c);
10 // Rise, fall and turn off delay
11 nor #(1,2,3) u_nor (out3,b,c);
12 //One Delay, min, typ and max
13 nand #(1:2:3) u_nand (out4,b,c);
14 //Two delays, min,typ and max
15 buf #(1:4:8,4:5:6) u_buf (out5,b);
16 //Three delays, min, typ, and max
17 notif1 #(1:2:3,4:5:6,7:8:9) u_notif1 (out6,b,c);
18
19 //Testbench code
20 initial begin
21   $monitor (
22     "Time=%g b=%b c=%b out1=%b out2=%b out3=%b out4=%b out5=%b out6=%b",
23     $time, b, c, out1, out2, out3, out4, out5, out6);
24   b = 0;
25   c = 0;
26   #10 b = 1;
27   #10 c = 1;
28   #10 b = 0;
29   #10 $finish;
30 end
31
32 endmodule
  
```

You could download file delay\_example.v [here](#)

```

Time = 0 b = 0 c=0 out1=x out2=x out3=x out4=x out5=x out6=x
Time = 1 b = 0 c=0 out1=x out2=x out3=1 out4=x out5=x out6=x
Time = 2 b = 0 c=0 out1=x out2=0 out3=1 out4=1 out5=x out6=z
Time = 5 b = 0 c=0 out1=0 out2=0 out3=1 out4=1 out5=0 out6=z
Time = 8 b = 0 c=0 out1=0 out2=0 out3=1 out4=1 out5=0 out6=z
Time = 10 b = 1 c=0 out1=0 out2=0 out3=1 out4=1 out5=0 out6=z
Time = 12 b = 1 c=0 out1=0 out2=0 out3=0 out4=1 out5=0 out6=z
Time = 14 b = 1 c=0 out1=0 out2=0 out3=0 out4=1 out5=1 out6=z
Time = 15 b = 1 c=0 out1=1 out2=0 out3=0 out4=1 out5=1 out6=z
Time = 20 b = 1 c=1 out1=1 out2=0 out3=0 out4=1 out5=1 out6=z
Time = 21 b = 1 c=1 out1=1 out2=1 out3=0 out4=1 out5=1 out6=z
Time = 22 b = 1 c=1 out1=1 out2=1 out3=0 out4=0 out5=1 out6=z
Time = 25 b = 1 c=1 out1=1 out2=1 out3=0 out4=0 out5=1 out6=0
Time = 30 b = 0 c=1 out1=1 out2=1 out3=0 out4=0 out5=1 out6=0
Time = 32 b = 0 c=1 out1=1 out2=0 out3=0 out4=1 out5=1 out6=1
Time = 35 b = 0 c=1 out1=1 out2=0 out3=0 out4=1 out5=0 out6=1
  
```

## N-Input Primitives

The and, nand, or, nor, xor, and xnor primitives have one output and any number of inputs.

- The single output is the first terminal.
- All other terminals are inputs.

## Examples

```

1 module n_in_primitive();
2
3 wire out1,out2,out3;
4 reg in1,in2,in3,in4;
5
6 // Two input AND gate
7 and u_and1 (out1, in1, in2);
8 // four input AND gate
9 and u_and2 (out2, in1, in2, in3, in4);
10 // three input XNOR gate
11 xnor u_xnor1 (out3, in1, in2, in3);
12
13 //Testbench Code
14 initial begin
15   $monitor (
16     "in1 = %b in2 = %b in3 = %b in4 = %b out1 = %b out2 = %b out3 = %b",
17     in1, in2, in3, in4, out1, out2, out3);
18   in1 = 0;
19   in2 = 0;
  
```

```

20    in3 = 0;
21    in4 = 0;
22    #1 in1 = 1;
23    #1 in2 = 1;
24    #1 in3 = 1;
25    #1 in4 = 1;
26    #1 $finish;
27 end
28
29 endmodule

```

You could download file n\_in\_primitive.v [here](#)

```

in1 = 0 in2 = 0 in3 = 0 in4 = 0 out1 = 0 out2 = 0 out3 = 1
in1 = 1 in2 = 0 in3 = 0 in4 = 0 out1 = 0 out2 = 0 out3 = 0
in1 = 1 in2 = 1 in3 = 0 in4 = 0 out1 = 1 out2 = 0 out3 = 1
in1 = 1 in2 = 1 in3 = 1 in4 = 0 out1 = 1 out2 = 0 out3 = 0
in1 = 1 in2 = 1 in3 = 1 in4 = 1 out1 = 1 out2 = 1 out3 = 0

```

## N-Output Primitives

The buf and not primitives have any number of outputs and one input

- The outputs are the first terminals listed.
- The last terminal is the single input.

## Examples

```

1 module n_out_primitive();
2
3 wire out,out_0,out_1,out_2,out_3,out_a,out_b,out_c;
4 wire in;
5
6 // one output Buffer gate
7 buf u_buf0 (out,in);
8 // four output Buffer gate
9 buf u_buf1 (out_0, out_1, out_2, out_3, in);
10 // three output Invertor gate
11 not u_not0 (out_a, out_b, out_c, in);
12
13 endmodule

```

You could download file n\_out\_primitive.v [here](#)

- Binary: +, -, \*, /, % (the modulus operator)
- Unary: +, - (This is used to specify the sign)
- Integer division truncates any fractional part
- The result of a modulus operation takes the sign of the first operand
- If any operand bit value is the unknown value x, then the entire result is x
- Register data types are used as unsigned values (Negative numbers stored in two's complement form)



### Example

```

1 module arithmetic_operators();
2
3 initial begin
4   $display (" 5 + 10 = %d", 5 + 10);
5   $display (" 5 - 10 = %d", 5 - 10);
6   $display (" 10 - 5 = %d", 10 - 5);
7   $display (" 10 * 5 = %d", 10 * 5);
8   $display (" 10 / 5 = %d", 10 / 5);
9   $display (" 10 / -5 = %d", 10 / -5);
10  $display (" 10 %s 3 = %d", "%", 10 % 3);
11  $display (" +5      = %d", +5);
12  $display (" -5      = %d", -5);
13  #10 $finish;
14 end
15
16 endmodule
  
```

You could download file arithmetic\_operators.v [here](#)

```

5 + 10 = 15
5 - 10 = -5
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2
10 / -5 = -2
10 % 3 = 1
+5      = 5
-5      = -5
  
```



### Relational Operators

| Operator | Description   |
|----------|---------------|
| $a < b$  | a less than b |

|            |                              |
|------------|------------------------------|
| $a > b$    | a greater than b             |
| $a \leq b$ | a less than or equal to b    |
| $a \geq b$ | a greater than or equal to b |

- The result is a scalar value (example  $a < b$ )
- 0 if the relation is false (a is bigger then b)
- 1 if the relation is true ( a is smaller then b)
- x if any of the operands has unknown x bits (if a or b contains X)

**Note:** If any operand is x or z, then the result of that test is treated as false (



### Example

```

1 module relational_operators();
2
3 initial begin
4   $display (" 5  <= 10 = %b", (5    <= 10));
5   $display (" 5  >= 10 = %b", (5    >= 10));
6   $display (" 1'bx <= 10 = %b", (1'bx <= 10));
7   $display (" 1'bz <= 10 = %b", (1'bz <= 10));
8   #10 $finish;
9 end
10
11 endmodule

```

You could download file relational\_operators.v [here](#)

```

5  <= 10 = 1
5  >= 10 = 0
1'bx <= 10 = x
1'bz <= 10 = x

```



### Equality Operators

There are two types of Equality operators. Case Equality and Logical Equality

| Operator  | Description  |
|-----------|--|
| $a === b$ | a equal to b, including x and z (Case equality)            |
| $a !== b$ | a not equal to b, including x and z (Case inequality)      |
| $a == b$  | a equal to b, result may be unknown (logical equality)     |
| $a != b$  | a not equal to b, result may be unknown (logical equality) |

- Operands are compared bit by bit, with zero filling if the two operands have the same length
- Result is 0 (false) or 1 (true)
- For the == and != operators, the result is x, if either operand contains z
- For the === and !== operators, bits with x and z are included in the comparison and must match for the result to be true

**Note :** The result is always 0 or 1.



### Example

```

1 module equality_operators();
2
3 initial begin
4   // Case Equality
5   $display (" 4'bx001 === 4'bx001 = %b", (4'bx001 === 4'bx001));
6   $display (" 4'bx0x1 === 4'bx001 = %b", (4'bx0x1 === 4'bx001));
7   $display (" 4'bz0x1 === 4'bz0x1 = %b", (4'bz0x1 === 4'bz0x1));
8   $display (" 4'bz0x1 === 4'bz001 = %b", (4'bz0x1 === 4'bz001));
9   // Case Inequality
10  $display (" 4'bx0x1 !== 4'bx001 = %b", (4'bx0x1 !== 4'bx001));
11  $display (" 4'bz0x1 !== 4'bz001 = %b", (4'bz0x1 !== 4'bz001));
12  // Logical Equality
13  $display (" 5 == 10 = %b", (5 == 10));
14  $display (" 5 == 5 = %b", (5 == 5));
15  // Logical Inequality
16  $display (" 5 != 5 = %b", (5 != 5));
17  $display (" 5 != 6 = %b", (5 != 6));
18  #10 $finish;
19 end
20
21 endmodule

```

You could download file equality\_operators.v [here](#)

```

4'bx001 === 4'bx001 = 1
4'bx0x1 === 4'bx001 = 0
4'bz0x1 === 4'bz0x1 = 1
4'bz0x1 === 4'bz001 = 0
4'bx0x1 !== 4'bx001 = 1
4'bz0x1 !== 4'bz001 = 1
5 == 10 = 0
5 == 5 = 1
5 != 5 = 0

```

```
5    != 6    = 1
```

## Logical Operators

| Operator | Description    |
|----------|----------------|
| !        | logic negation |
| &&       | logical and    |
|          | logical or     |

- Expressions connected by && and || are evaluated from left to right
- Evaluation stops as soon as the result is known
- The result is a scalar value:
  - 0 if the relation is false
  - 1 if the relation is true
  - x if any of the operands has x (unknown) bits



## Example

```

1  module logical_operators();
2
3  initial begin
4    // Logical AND
5    $display ("1'b1 && 1'b1 = %b", (1'b1 && 1'b1));
6    $display ("1'b1 && 1'b0 = %b", (1'b1 && 1'b0));
7    $display ("1'b1 && 1'bx = %b", (1'b1 && 1'bx));
8    // Logical OR
9    $display ("1'b1 || 1'b0 = %b", (1'b1 || 1'b0));
10   $display ("1'b0 || 1'b0 = %b", (1'b0 || 1'b0));
11   $display ("1'b0 || 1'bx = %b", (1'b0 || 1'bx));
12   // Logical Negation
13   $display ("! 1'b1    = %b", (! 1'b1));
14   $display ("! 1'b0    = %b", (! 1'b0));
15   #10 $finish;
16 end
17
18 endmodule

```

You could download file logical\_operators.v [here](#)

```

1'b1 && 1'b1 = 1
1'b1 && 1'b0 = 0
1'b1 && 1'bx = x
1'b1 || 1'b0 = 1

```

```

1'b0 || 1'b0 = 0
1'b0 || 1'bx = x
! 1'b1      = 0
! 1'b0      = 1
  
```

## Bit-wise Operators

Bitwise operators perform a bit wise operation on two operands. They take one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be extended on the left with zeroes to match the length of the longer operand.

| Operator | Description                 |
|----------|-----------------------------|
| ~        | negation                    |
| &        | and                         |
|          | inclusive or                |
| ^        | exclusive or                |
| ^~ or ~^ | exclusive nor (equivalence) |

- Computations include unknown bits, in the following way:
  - $\sim x = x$
  - $0 \& x = 0$
  - $1 \& x = x \& x = x$
  - $1 | x = 1$
  - $0 | x = x | x = x$
  - $0 \wedge x = 1 \wedge x = x \wedge x = x$
  - $0 \wedge \sim x = 1 \wedge \sim x = x \wedge \sim x = x$
- When operands are of unequal bit length, the shorter operand is zero extended to the most significant bit positions.



## Example

```

1 module bitwise_operators();
2
3 initial begin
4   // Bit Wise Negation
5   $display (" ~4'b0001 = %b", (~4'b0001));
6   $display (" ~4'bx001 = %b", (~4'bx001));
7   $display (" ~4'bz001 = %b", (~4'bz001));
8   // Bit Wise AND
9   $display (" 4'b0001 & 4'b1001 = %b", (4'b0001 & 4'b1001));
  
```



## Verilog Programming Guide

```

10 $display (" 4'b1001 & 4'bx001 = %b", (4'b1001 & 4'bx001));
11 $display (" 4'b1001 & 4'bz001 = %b", (4'b1001 & 4'bz001));
12 // Bit Wise OR
13 $display (" 4'b0001 | 4'b1001 = %b", (4'b0001 | 4'b1001));
14 $display (" 4'b0001 | 4'bx001 = %b", (4'b0001 | 4'bx001));
15 $display (" 4'b0001 | 4'bz001 = %b", (4'b0001 | 4'bz001));
16 // Bit Wise XOR
17 $display (" 4'b0001 ^ 4'b1001 = %b", (4'b0001 ^ 4'b1001));
18 $display (" 4'b0001 ^ 4'bx001 = %b", (4'b0001 ^ 4'bx001));
19 $display (" 4'b0001 ^ 4'bz001 = %b", (4'b0001 ^ 4'bz001));
20 // Bit Wise XNOR
21 $display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001));
22 $display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001));
23 $display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001));
24 #10 $finish;
25 end
26
27 endmodule

```

You could download file bitwise\_operators.v [here](#)

```

~4'b0001      = 1110
~4'bx001      = x110
~4'bz001      = x110
4'b0001 & 4'b1001 = 0001
4'b1001 & 4'bx001 = x001
4'b1001 & 4'bz001 = x001
4'b0001 | 4'b1001 = 1001
4'b0001 | 4'bx001 = x001
4'b0001 | 4'bz001 = x001
4'b0001 ^ 4'b1001 = 1000
4'b0001 ^ 4'bx001 = x000
4'b0001 ^ 4'bz001 = x000
4'b0001 ~^ 4'b1001 = 0111
4'b0001 ~^ 4'bx001 = x111
4'b0001 ~^ 4'bz001 = x111

```

## Reduction Operators

| Operator | Description |
|----------|-------------|
| &        | and         |
| ~&       | nand        |
|          | or          |
| ~        | nor         |
| ^        | xor         |

$\wedge \sim$  or  $\sim \wedge$ 

xnor

- Reduction operators are unary.
- They perform a bit-wise operation on a single operand to produce a single bit result.
- Reduction unary NAND and NOR operators operate as AND and OR respectively.
  - Unknown bits are treated as described before.

### Example

```

1 module reduction_operators();
2
3 initial begin
4   // Bit Wise AND reduction
5   $display (" & 4'b1001 = %b", (& 4'b1001));
6   $display (" & 4'bx111 = %b", (& 4'bx111));
7   $display (" & 4'bz111 = %b", (& 4'bz111));
8   // Bit Wise NAND reduction
9   $display (" ~& 4'b1001 = %b", (~& 4'b1001));
10  $display (" ~& 4'bx001 = %b", (~& 4'bx001));
11  $display (" ~& 4'bz001 = %b", (~& 4'bz001));
12  // Bit Wise OR reduction
13  $display (" | 4'b1001 = %b", (| 4'b1001));
14  $display (" | 4'bx000 = %b", (| 4'bx000));
15  $display (" | 4'bz000 = %b", (| 4'bz000));
16  // Bit Wise NOR reduction
17  $display (" ~| 4'b1001 = %b", (~| 4'b1001));
18  $display (" ~| 4'bx001 = %b", (~| 4'bx001));
19  $display (" ~| 4'bz001 = %b", (~| 4'bz001));
20  // Bit Wise XOR reduction
21  $display (" ^ 4'b1001 = %b", (^ 4'b1001));
22  $display (" ^ 4'bx001 = %b", (^ 4'bx001));
23  $display (" ^ 4'bz001 = %b", (^ 4'bz001));
24  // Bit Wise XNOR
25  $display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
26  $display (" ~^ 4'bx001 = %b", (~^ 4'bx001));
27  $display (" ~^ 4'bz001 = %b", (~^ 4'bz001));
28  #10 $finish;
29 end
30
31 endmodule

```

You could download file reduction\_operators.v [here](#)

```

& 4'b1001 = 0
& 4'bx111 = x

```

```

& 4'bz111 = x
~& 4'b1001 = 1
~& 4'bx001 = 1
~& 4'bz001 = 1
| 4'b1001 = 1
| 4'bx000 = x
| 4'bz000 = x
~| 4'b1001 = 0
~| 4'bx001 = 0
~| 4'bz001 = 0
^ 4'b1001 = 0
^ 4'bx001 = x
^ 4'bz001 = x
~^ 4'b1001 = 1
~^ 4'bx001 = x
~^ 4'bz001 = x
  
```

## ● Shift Operators

| Operator | Description |
|----------|-------------|
| <<       | left shift  |
| >>       | right shift |

- The left operand is shifted by the number of bit positions given by the right operand.
- The vacated bit positions are filled with zeroes.

## ◆ Example

```

1 module shift_operators();
2
3 initial begin
4   // Left Shift
5   $display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
6   $display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));
7   $display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
8   // Right Shift
9   $display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
10  $display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
11  $display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
12  #10 $finish;
13 end
14
15 endmodule
  
```

You could download file shift\_operators.v [here](#)

```

4'b1001 << 1 = 0010
4'b10x1 << 1 = 0x10
4'b10z1 << 1 = 0z10
4'b1001 >> 1 = 0100
4'b10x1 >> 1 = 010x
4'b10z1 >> 1 = 010z
  
```

## Concatenation Operator

- Concatenations are expressed using the brace characters { and }, with commas separating the operands.
  - Example: + {a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the result is a 12-bit number.
- Unsize constant numbers are not allowed in concatenations.

## Example

```

1 module concatenation_operator();
2
3 initial begin
4   // concatenation
5   $display (" {4'b1001,4'b10x1} = %b", {4'b1001,4'b10x1});
6   #10 $finish;
7 end
8
9 endmodule
  
```

You could download file concatenation\_operator.v [here](#)

```
{4'b1001,4'b10x1} = 100110x1
```

## Replication Operator

Replication operator is used to replicate a group of bits n times. Say you have a 4 bit value, you can use the replication operator.

| Operator | Description                |
|----------|----------------------------|
| {n{m}}   | Replicate value m, n times |

- Repetition multipliers (must be constants) can be used:
  - {3{a}} // this is equivalent to {a, a, a}
- Nested concatenations and replication operator are possible:

- {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

### Example

```

1 module replication_operator();
2
3 initial begin
4   // replication
5   $display (" {4{4'b1001}} = %b", {4{4'b1001}});
6   // replication and concatenation
7   $display (" {4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});
8   #10 $finish;
9 end
10
11 endmodule

```

You could download file replication\_operator.v [here](#)

```

{4{4'b1001}} = 1001100110011001
{4{4'b1001,1'bz}} = 1001z1001z1001z1001z

```

## Conditional Operators

- The conditional operator has the following C-like format:
  - cond\_expr ? true\_expr : false\_expr
- The true\_expr or the false\_expr is evaluated and used as a result depending on v

### Example

```

1 module conditional_operator();
2
3 wire out;
4 reg enable,data;
5 // Tri state buffer
6 assign out = (enable) ? data : 1'bz;
7
8 initial begin
9   $display ("time\t enable data out");
10  $monitor ("%g\t %b   %b   %b", $time, enable, data, out);
11  enable = 0;
12  data = 0;
13  #1 data = 1;

```

```

14  #1 data = 0;
15  #1 enable = 1;
16  #1 data = 1;
17  #1 data = 0;
18  #1 enable = 0;
19  #10 $finish;
20  end
21
22  endmodule
  
```

You could download file conditional\_operator.v [here](#)

| time | enable | data | out |
|------|--------|------|-----|
| 0    | 0      | 0    | z   |
| 1    | 0      | 1    | z   |
| 2    | 0      | 0    | z   |
| 3    | 1      | 0    | 0   |
| 4    | 1      | 1    | 1   |
| 5    | 1      | 0    | 0   |
| 6    | 0      | 0    | z   |

## Operator Precedence

| Operator                         | Symbols                  |
|----------------------------------|--------------------------|
| Unary, Multiply, Divide, Modulus | !, ~, *, /, %            |
| Add, Subtract, Shift             | +, -, <<, >>             |
| Relation, Equality               | <, >, <=, >=, ==, !=, == |
| Reduction                        | &, !&, ^, ^~,  , ~       |
| Logic                            | &&,                      |
| Conditional                      | ? :                      |

## Verilog HDL Abstraction Levels

- Behavioral Models : Higher level of modeling where behavior logic is modeled.
- RTL Models : Logic is modeled at register level
- Structural Models : Logic is modeled at both register level and gate level.



## Procedural Blocks

Verilog behavioral code is inside procedure blocks, but there is an exception: some behavioral code also exists outside procedure blocks. You can see this in detail as we make progress.

There are two types of procedural blocks in Verilog:

- **initial** : initial blocks execute only once at time zero (start execution at time zero).
- **always** : always blocks loop to execute over and over again. In other words, as the name suggests, it executes always.



### Example - initial

```
1 module initial_example();  
2 reg clk,reset,enable,data;  
3  
4 initial begin  
5     clk = 0;  
6     reset = 0;  
7     enable = 0;  
8     data = 0;  
9 end  
10  
11 endmodule
```

You could download file initial\_example.v [here](#)

In the above example, the initial block execution and always block execution starts at time 0. Always block waits for the event, here positive edge of clock, whereas initial block just executed all the statements between begin and end statement, without waiting.



### Example - always

```
1 module always_example();  
2 reg clk,reset,enable,q_in,data;  
3  
4 always @ (posedge clk)  
5 if (reset) begin  
6     data <= 0;  
7 end else if (enable) begin  
8     data <= q_in;  
9 end
```

```
10  
11 endmodule
```

You could download file `always_example.v` [here](#)

In an `always` block, when the trigger event occurs, the code inside and end is executed; then once again the `always` block waits for next triggering. This process of waiting and executing on event is repeated until simulation stops.



## Procedural Assignment Statements

- Procedural assignment statements assign values to `reg`, `integer`, `real`, or `time` variables and can not assign values to `nets` (wires, `io` types)
- You can assign to a register (`reg` data type) the value of a `net` (wire), constant, another register, or a specific value.



## Example - Bad procedural assignment

```
1 module initial_bad();  
2 reg clk, reset;  
3 wire enable, data;  
4  
5 initial begin  
6     clk = 0;  
7     reset = 0;  
8     enable = 0;  
9     data = 0;  
10 end  
11  
12 endmodule
```

You could download file `initial_bad.v` [here](#)



## Example - Good procedural assignment

```
1 module initial_good();  
2 reg clk, reset, enable, data;  
3  
4 initial begin  
5     clk = 0;  
6     reset = 0;  
7     enable = 0;  
8     data = 0;
```



```

9 end
10
11 endmodule

```

You could download file initial\_good.v [here](#)



## Procedural Assignment Groups

If a procedure block contains more than one statement, those statements must be enclosed within

- Sequential **begin - end** block
- Parallel **fork - join** block

When using begin-end, we can give name to that group. These are called **named blocks**.



## Example - "begin-end"

```

1 module initial_begin_end();
2   reg clk, reset, enable, data;
3
4   initial begin
5     $monitor (
6       "%g clk=%b reset=%b enable=%b data=%b",
7       $time, clk, reset, enable, data);
8     #1   clk = 0;
9     #10  reset = 0;
10    #5   enable = 0;
11    #3   data = 0;
12    #1   $finish;
13  end
14
15 endmodule

```

You could download file initial\_begin\_end.v [here](#)

**Begin** : clk gets 0 after 1 time unit, reset gets 0 after 11 time units, after 16 time units, data after 19 units. All the statements are executed sequentially.

## Simulator Output

```

0 clk=x reset=x enable=x data=x
1 clk=0 reset=x enable=x data=x
11 clk=0 reset=0 enable=x data=x
16 clk=0 reset=0 enable=0 data=x

```

```
19 clk=0 reset=0 enable=0 data=0
```



### Example - "fork-join"

```

1 module initial_fork_join();
2 reg clk,reset,enable,data;
3
4 initial begin
5   $monitor ("%g clk=%b reset=%b enable=%b data=%b",
6     $time, clk, reset, enable, data);
7   fork
8     #1   clk = 0;
9     #10  reset = 0;
10    #5   enable = 0;
11    #3   data = 0;
12   join
13   #1 $display ("%g Terminating simulation", $time);
14   $finish;
15 end
16
17 endmodule

```

You could download file initial\_fork\_join.v [here](#)

**Fork** : clk gets its value after 1 time unit, reset after 10 time units, after 5 time units, data after 3 time units. All the statements are executed in parallel.

### Simulator Output

```

0 clk=x reset=x enable=x data=x
1 clk=0 reset=x enable=x data=x
3 clk=0 reset=x enable=x data=0
5 clk=0 reset=x enable=0 data=0
10 clk=0 reset=0 enable=0 data=0
11 Terminating simulation

```



### Sequential Statement Groups

The **begin - end** keywords:

- Group several statements together.
- Cause the statements to be evaluated sequentially (one at a time).

- Any timing within the sequential groups is relative to previous statement.
- Delays in the sequence accumulate (each delay is added to the previous delay)
- Block finishes after the last statement in the block.



### Example - sequential

```
1 module sequential();  
2  
3 reg a;  
4  
5 initial begin  
6     $monitor ("%g a = %b", $time, a);  
7     #10 a = 0;  
8     #11 a = 1;  
9     #12 a = 0;  
10    #13 a = 1;  
11    #14 $finish;  
12 end  
13  
14 endmodule
```

You could download file sequential.v [here](#)

### Simulator Output

```
0 a = x  
10 a = 0  
21 a = 1  
33 a = 0  
46 a = 1
```



### Parallel Statement Groups

The fork - join keywords:

- Group several statements together.
- Cause the statements to be evaluated in parallel (all at the same time).
  - Timing within parallel group is absolute to the beginning of the group.

- Block finishes after the last statement completes (Statement with highest delay, it can be the first state in the block).



### Example - Parallel

```
1 module parallel();
2
3 reg a;
4
5 initial
6 fork
7     $monitor ("%g a = %b", $time, a);
8     #10 a = 0;
9     #11 a = 1;
10    #12 a = 0;
11    #13 a = 1;
12    #14 $finish;
13 join
14
15 endmodule
```

You could download file parallel.v [here](#)

### Simulator Output

```
0 a = x
10 a = 0
11 a = 1
12 a = 0
13 a = 1
```



### Example - Mixing "begin-end" and "fork - join"

```
1 module fork_join();
2
3 reg clk, reset, enable, data;
4
5 initial begin
6     $display ("Starting simulation");
7     $monitor ("%g clk=%b reset=%b enable=%b data=%b",
8         $time, clk, reset, enable, data);
9     fork : FORK_VAL
10        #1 clk = 0;
```

```
11      #5  reset = 0;
12      #5  enable = 0;
13      #2  data = 0;
14      join
15      #10 $display ("%g Terminating simulation", $time);
16      $finish;
17  end
18
19  endmodule
```

You could download file fork\_join.v [here](#)

## Simulator Output

```
0 clk=x reset=x enable=x data=x
1 clk=0 reset=x enable=x data=x
2 clk=0 reset=x enable=x data=0
5 clk=0 reset=0 enable=0 data=0
15 Terminating simulation
```



## Blocking and Nonblocking assignment

Blocking assignments are executed in the order they are coded, they are sequential. Since they block the execution of next statement, current statement is executed, they are called blocking assignment. Assignment are made with "=" symbol. Example a = b;

Nonblocking assignments are executed in parallel. Since the execution of next statement is not blocked due to execution of current statement, they are called nonblocking statement. Assignments are made with "<=" symbol. Example a <= b;

**Note :** Correct way to spell 'nonblocking' is 'nonblocking' and not 'non blocking'.



## Example - blocking and nonblocking

```
1  module blocking_nonblocking();
2
3  reg a,b,c,d;
4  // Blocking Assignment
5  initial begin
6      #10 a = 0;
7      #11 a = 1;
8      #12 a = 0;
9      #13 a = 1;
```

```

10 end
11
12 initial begin
13     #10 b <= 0;
14     #11 b <= 1;
15     #12 b <= 0;
16     #13 b <= 1;
17 end
18
19 initial begin
20     c = #10 0;
21     c = #11 1;
22     c = #12 0;
23     c = #13 1;
24 end
25
26 initial begin
27     d <= #10 0;
28     d <= #11 1;
29     d <= #12 0;
30     d <= #13 1;
31 end
32
33 initial begin
34     $monitor("TIME = %g A = %b B = %b C = %b D = %b", $time, a,
d);
35     #50 $finish;
36 end
37
38 endmodule
  
```

You could download file blocking\_nonblocking.v [here](#)

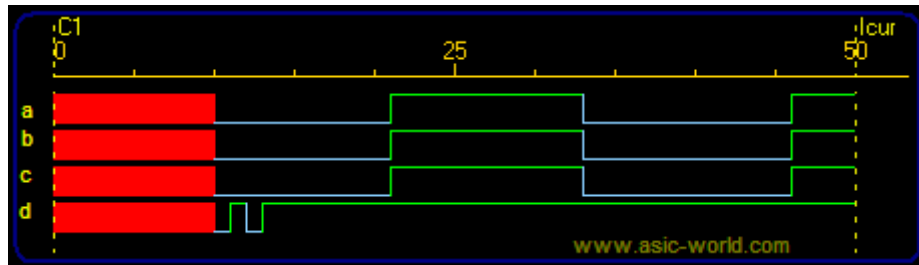
## Simulator Output

```

TIME = 0 A = x B = x C = x D = x
TIME = 10 A = 0 B = 0 C = 0 D = 0
TIME = 11 A = 0 B = 0 C = 0 D = 1
TIME = 12 A = 0 B = 0 C = 0 D = 0
TIME = 13 A = 0 B = 0 C = 0 D = 1
TIME = 21 A = 1 B = 1 C = 1 D = 1
TIME = 33 A = 0 B = 0 C = 0 D = 1
TIME = 46 A = 1 B = 1 C = 1 D = 1
  
```



## Waveform



### assign and deassign

The assign and deassign procedural assignment statements continuous assignments to be placed onto registers for controlled of time. The assign procedural statement overrides previous assignments to a **register**. The deassign procedural statement removes a continuous assignment to a register.



### Example - assign and deassign

```

1 module assign_deassign ();
2
3 reg clk,rst,d,preset;
4 wire q;
5
6 initial begin
7   $monitor ("%g clk %b rst %b preset %b d %b q %b",
8     $time, clk, rst, preset, d, q);
9   clk = 0;
10  rst = 0;
11  d = 0;
12  preset = 0;
13  #10 rst = 1;
14  #10 rst = 0;
15  repeat (10) begin
16    @ (posedge clk);
17    d <= $random;
18    @ (negedge clk);
19    preset <= ~preset;
20  end
21  #1 $finish;
22 end
23 // Clock generator
24 always #1 clk = ~clk;
25
26 // assign and deassign q of flip flop module
27 always @(preset)
28 if (preset) begin

```

```

29    assign U.q = 1; // assign procedural statement
30  end else begin
31    deassign U.q;    // deassign procedural statement
32  end
33
34  d_ff U (clk,rst,d,q);
35
36  endmodule
37
38  // D Flip-Flop model
39  module d_ff (clk,rst,d,q);
40  input  clk,rst,d;
41  output q;
42  reg q;
43
44  always @ (posedge clk)
45  if (rst) begin
46    q <= 0;
47  end else begin
48    q <= d;
49  end
50
51  endmodule

```

You could download file assign\_deassign.v [here](#)

## Simulator Output

```

@0 clk 0 rst 0 preset 0 d 0 q x
@1 clk 1 rst 0 preset 0 d 0 q 0
@2 clk 0 rst 0 preset 0 d 0 q 0
@3 clk 1 rst 0 preset 0 d 0 q 0
@4 clk 0 rst 0 preset 0 d 0 q 0
@5 clk 1 rst 0 preset 0 d 0 q 0
@6 clk 0 rst 0 preset 0 d 0 q 0
@7 clk 1 rst 0 preset 0 d 0 q 0
@8 clk 0 rst 0 preset 0 d 0 q 0
@9 clk 1 rst 0 preset 0 d 0 q 0
@10 clk 0 rst 1 preset 0 d 0 q 0
@11 clk 1 rst 1 preset 0 d 0 q 0
@12 clk 0 rst 1 preset 0 d 0 q 0
@13 clk 1 rst 1 preset 0 d 0 q 0
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1

```



```

@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 1
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 1
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
  
```



### force and release

Another form of procedural continuous assignment is provided by the force and release procedural statements. These statements have a similar syntax to the assign-deassign pair, but a force can be applied to nets as well as to registers.

One can use force and release while doing gate level simulation to simulate around reset connectivity problems. Also can be used to insert single or double bit errors on data read from memory.



### Example - force and release

```

1 module force_release ();
2
3 reg clk,rst,d,preset;
4 wire q;
5
6 initial begin
7   $monitor ("%g clk %b rst %b preset %b d %b q %b",
8     $time, clk, rst, preset, d, q);
9   clk = 0;
10  rst = 0;
11  d = 0;
12  preset = 0;
13  #10 rst = 1;
14  #10 rst = 0;
  
```

```

15  repeat (10) begin
16      @ (posedge clk);
17      d <= $random;
18      @ (negedge clk) ;
19      preset <= ~preset;
20  end
21  #1 $finish;
22 end
23 // Clock generator
24 always #1 clk = ~clk;
25
26 // force and release of flip flop module
27 always @(preset)
28 if (preset) begin
29     force U.q = preset; // force procedural statement
30 end else begin
31     release U.q; // release procedural statement
32 end
33
34 d_ff U (clk,rst,d,q);
35
36 endmodule
37
38 // D Flip-Flop model
39 module d_ff (clk,rst,d,q);
40 input clk,rst,d;
41 output q;
42 wire q;
43 reg q_reg;
44
45 assign q = q_reg;
46
47 always @ (posedge clk)
48 if (rst) begin
49     q_reg <= 0;
50 end else begin
51     q_reg <= d;
52 end
53
54 endmodule
  
```

You could download file force\_release.v [here](#)

## Simulator Output

```

@0 clk 0 rst 0 preset 0 d 0 q x
@1 clk 1 rst 0 preset 0 d 0 q 0
@2 clk 0 rst 0 preset 0 d 0 q 0
@3 clk 1 rst 0 preset 0 d 0 q 0
  
```

```

@4  clk 0 rst 0 preset 0 d 0 q 0
@5  clk 1 rst 0 preset 0 d 0 q 0
@6  clk 0 rst 0 preset 0 d 0 q 0
@7  clk 1 rst 0 preset 0 d 0 q 0
@8  clk 0 rst 0 preset 0 d 0 q 0
@9  clk 1 rst 0 preset 0 d 0 q 0
@10 clk 0 rst 1 preset 0 d 0 q 0
@11 clk 1 rst 1 preset 0 d 0 q 0
@12 clk 0 rst 1 preset 0 d 0 q 0
@13 clk 1 rst 1 preset 0 d 0 q 0
@14 clk 0 rst 1 preset 0 d 0 q 0
@15 clk 1 rst 1 preset 0 d 0 q 0
@16 clk 0 rst 1 preset 0 d 0 q 0
@17 clk 1 rst 1 preset 0 d 0 q 0
@18 clk 0 rst 1 preset 0 d 0 q 0
@19 clk 1 rst 1 preset 0 d 0 q 0
@20 clk 0 rst 0 preset 0 d 0 q 0
@21 clk 1 rst 0 preset 0 d 0 q 0
@22 clk 0 rst 0 preset 1 d 0 q 1
@23 clk 1 rst 0 preset 1 d 1 q 1
@24 clk 0 rst 0 preset 0 d 1 q 0
@25 clk 1 rst 0 preset 0 d 1 q 1
@26 clk 0 rst 0 preset 1 d 1 q 1
@27 clk 1 rst 0 preset 1 d 1 q 1
@28 clk 0 rst 0 preset 0 d 1 q 1
@29 clk 1 rst 0 preset 0 d 1 q 1
@30 clk 0 rst 0 preset 1 d 1 q 1
@31 clk 1 rst 0 preset 1 d 1 q 1
@32 clk 0 rst 0 preset 0 d 1 q 1
@33 clk 1 rst 0 preset 0 d 1 q 1
@34 clk 0 rst 0 preset 1 d 1 q 1
@35 clk 1 rst 0 preset 1 d 0 q 1
@36 clk 0 rst 0 preset 0 d 0 q 1
@37 clk 1 rst 0 preset 0 d 1 q 0
@38 clk 0 rst 0 preset 1 d 1 q 1
@39 clk 1 rst 0 preset 1 d 1 q 1
@40 clk 0 rst 0 preset 0 d 1 q 1
  
```

**The  
Conditional  
Statement if-  
else**

























## Looping Statements











## Continuous Assignment Statements

Continuous assignment statements drive nets (wire data type). They represent structural connections.

- They are used for modeling Tri-State buffers.
- They can be used for modeling combinational logic.
- They are outside the procedural blocks (always and initial blocks).
- The continuous assign overrides any procedural assignments.
- The left-hand side of a continuous assignment must be net data type.

**syntax :** assign (strength, strength) **#(delay)** net = expression;



### Example - One bit Adder

```
1 module adder_using_assign ();
2 reg a, b;
3 wire sum, carry;
4
5 assign #5 {carry,sum} = a+b;
6
7 initial begin
8     $monitor (" A = %b B = %b CARRY = %b SUM =
9     %b", a,b,carry,sum) ;
10    #10 a = 0;
11    b = 0;
12    #10 a = 1;
13    #10 b = 1;
14    #10 a = 0;
15    #10 b = 0;
16    #10 $finish;
17 end
18 endmodule
```

You could download file adder\_using\_assign.v [here](#)



### Example - Tri-state buffer

```
1 module tri_buf_using_assign();
```

```

2 reg data_in, enable;
3 wire pad;
4
5 assign pad = (enable) ? data_in : 1'bz;
6
7 initial begin
8     $monitor ("TIME = %g ENABLE = %b DATA : %b PAD %b",
9         $time, enable, data_in, pad);
10    #1 enable = 0;
11    #1 data_in = 1;
12    #1 enable = 1;
13    #1 data_in = 0;
14    #1 enable = 0;
15    #1 $finish;
16 end
17
18 endmodule

```

You could download file tri\_buf\_using\_assign.v [here](#)



### Propagation Delay

Continuous Assignments may have a delay specified; only one delay for all transitions may be specified. A minimum:typical:maximum delay range may be specified.



### Example - Tri-state buffer

```

1 module tri_buf_using_assign_delays();
2 reg data_in, enable;
3 wire pad;
4
5 assign #(1:2:3) pad = (enable) ? data_in : 1'bz;
6
7 initial begin
8     $monitor ("ENABLE = %b DATA : %b PAD %b", enable,
9         data_in, pad);
10    #10 enable = 0;
11    #10 data_in = 1;
12    #10 enable = 1;
13    #10 data_in = 0;
14    #10 enable = 0;
15    #10 $finish;
16 end
17 endmodule

```

You could download file tri\_buf\_using\_assign\_delays.v [here](#)

## Procedural Block Control

Procedural blocks become active at simulation time zero. Use level sensitive event controls to control the execution of a procedure.

```
1 module dlatch_using_always();
2 reg q;
3
4 reg d, enable;
5
6 always @ (d or enable)
7 if (enable) begin
8     q = d;
9 end
10
11 initial begin
12     $monitor (" ENABLE = %b D = %b Q = %b", enable, d, q);
13     #1 enable = 0;
14     #1 d = 1;
15     #1 enable = 1;
16     #1 d = 0;
17     #1 d = 1;
18     #1 d = 0;
19     #1 enable = 0;
20     #10 $finish;
21 end
22
23 endmodule
```

You could download file dlatch\_using\_always.v [here](#)

Any change in either d or enable satisfies the event control and allows the execution of the statements in the procedure. The procedure is sensitive to any change in d or enable.



## Combo Logic using Procedural Coding

To model combinational logic, a procedure block must be sensitive to any change on the input. There is one important rule that needs to be followed while modeling combinational logic. If you use conditional checking using "if", then you need to mention the "else" part. Missing the else part results in a latch. If you don't like typing the else part, then you must initialize all the variables of that combo block as soon as it enters.



## Example - One bit Adder

```

1 module adder_using_always ();
2 reg a, b;
3 reg sum, carry;
4
5 always @ (a or b)
6 begin
7   {carry,sum} = a + b;
8 end
9
10 initial begin
11   $monitor (" A = %b B = %b CARRY = %b SUM =
12   %b",a,b,carry,sum);
13   #10 a = 0;
14   b = 0;
15   #10 a = 1;
16   #10 b = 1;
17   #10 a = 0;
18   #10 b = 0;
19   #10 $finish;
20 end
21 endmodule

```

You could download file adder\_using\_always.v [here](#)

The statements within the procedural block work with entire vectors at a time.



### Example - 4-bit Adder

```

1 module adder_4_bit_using_always ();
2 reg[3:0] a, b;
3 reg [3:0] sum;
4 reg carry;
5
6 always @ (a or b)
7 begin
8   {carry,sum} = a + b;
9 end
10
11 initial begin
12   $monitor (" A = %b B = %b CARRY = %b SUM =
13   %b",a,b,carry,sum);
14   #10 a = 8;
15   b = 7;
16   #10 a = 10;
17   #10 b = 15;
18   #10 a = 0;

```

```
18     #10 b = 0;
19     #10 $finish;
20 end
21
22 endmodule
```

You could download file `adder_4_bit_using_always.v` [here](#)



### Example - Ways to avoid Latches - Cover all conditions

```
1 module avoid_latch_else ();
2
3 reg q;
4 reg enable, d;
5
6 always @ (enable or d)
7 if (enable) begin
8     q = d;
9 end else begin
10    q = 0;
11 end
12
13 initial begin
14     $monitor (" ENABLE = %b D = %b Q = %b", enable, d, q);
15     #1 enable = 0;
16     #1 d = 0;
17     #1 enable = 1;
18     #1 d = 1;
19     #1 d = 0;
20     #1 d = 1;
21     #1 d = 0;
22     #1 d = 1;
23     #1 enable = 0;
24     #1 $finish;
25 end
26
27 endmodule
```

You could download file `avoid_latch_else.v` [here](#)



### Example - Ways to avoid Latches - Snit the variables to zero

```
1 module avoid_latch_init ();
2
3 reg q;
4 reg enable, d;
5
6 always @ (enable or d)
```



```
7 begin
8     q = 0;
9     if (enable) begin
10         q = d;
11     end
12 end
13
14 initial begin
15     $monitor (" ENABLE = %b D = %b Q = %b", enable, d, q);
16     #1 enable = 0;
17     #1 d = 0;
18     #1 enable = 1;
19     #1 d = 1;
20     #1 d = 0;
21     #1 d = 1;
22     #1 d = 0;
23     #1 d = 1;
24     #1 enable = 0;
25     #1 $finish;
26 end
27
28 endmodule
```

You could download file `avoid_latch_init.v` [here](#)



### Sequential Logic using Procedural Coding

To model sequential logic, a procedure block must be sensitive to positive edge or negative edge of clock. To model asynchronous reset, procedure block must be sensitive to both clock and reset. All the assignments to sequential logic should be made through nonblocking assignments.

Sometimes it's tempting to have multiple edge triggering variables in the sensitive list: this is fine for simulation. But for synthesis this does not make sense, as in real life, flip-flop can have only one clock, one reset and one preset (i.e. posedge clk or posedge reset or posedge preset).

One common mistake the new beginner makes is using clock as the enable input to flip-flop. This is fine for simulation, but for synthesis, this is not right.



### Example - Bad coding - Using two clocks

```
1 module wrong_seq();
2
3 reg q;
```

```

4 reg clk1, clk2, d1, d2;
5
6 always @ (posedge clk1 or posedge clk2)
7 if (clk1) begin
8   q <= d1;
9 end else if (clk2) begin
10  q <= d2;
11 end
12
13 initial begin
14   $monitor ("CLK1 = %b CLK2 = %b D1 = %b D2 %b Q = %b",
15     clk1, clk2, d1, d2, q);
16   clk1 = 0;
17   clk2 = 0;
18   d1 = 0;
19   d2 = 1;
20   #10 $finish;
21 end
22
23 always
24   #1 clk1 = ~clk1;
25
26 always
27   #1.9 clk2 = ~clk2;
28
29 endmodule

```

You could download file wrong\_seq.v [here](#)



### Example - D Flip-flop with async reset and async preset

```

1 module dff_async_reset_async_preset();
2
3 reg clk, reset, preset, d;
4 reg q;
5
6 always @ (posedge clk or posedge reset or posedge preset)
7 if (reset) begin
8   q <= 0;
9 end else if (preset) begin
10  q <= 1;
11 end else begin
12  q <= d;
13 end
14
15 // Testbench code here
16 initial begin
17   $monitor ("CLK = %b RESET = %b PRESET = %b D = %b Q = %b",

```

```

18      clk,reset,preset,d,q) ;
19      clk      = 0;
20      #1 reset  = 0;
21      preset = 0;
22      d        = 0;
23      #1 reset = 1;
24      #2 reset = 0;
25      #2 preset = 1;
26      #2 preset = 0;
27      repeat (4) begin
28          #2 d      = ~d;
29      end
30      #2 $finish;
31 end
32
33 always
34     #1 clk = ~clk;
35
36 endmodule
  
```

You could download file dff\_async\_reset\_async\_preset.v [here](#)



### Example - D Flip-flop with sync reset and sync preset

```

1 module dff_sync_reset_sync_preset() ;
2
3 reg clk,reset,preset,d;
4 reg q;
5
6 always @ (posedge clk)
7 if (reset) begin
8     q <= 0;
9 end else if (preset) begin
10    q <= 1;
11 end else begin
12    q <= d;
13 end
14
15 // Testbench code here
16 initial begin
17     $monitor("CLK = %b RESET = %b PRESET = %b D = %b Q = %b",
18         clk,reset,preset,d,q) ;
19     clk      = 0;
20     #1 reset  = 0;
21     preset = 0;
22     d        = 0;
23     #1 reset = 1;
24     #2 reset = 0;
  
```

```
25  #2 preset = 1;
26  #2 preset = 0;
27  repeat (4) begin
28      #2 d      = ~d;
29  end
30  #2 $finish;
31 end
32
33 always
34 #1 clk = ~clk;
35
36 endmodule
```

You could download file dff\_sync\_reset\_sync\_preset.v [here](#)



### A procedure can't trigger itself

One cannot trigger the block with a variable that block assigns value or drives.

```
1  module trigger_itself();
2
3  reg clk;
4
5  always @ (clk)
6      #5 clk = !clk;
7
8  // Testbench code here
9  initial begin
10     $monitor("TIME = %d CLK = %b", $time, clk);
11     clk = 0;
12     #500 $display("TIME = %d CLK = %b", $time, clk);
13     $finish;
14 end
15
16 endmodule
```

You could download file trigger\_itself.v [here](#)



### Procedural Block Concurrency

If we have multiple always blocks inside one module, then all the blocks (i.e. all the always blocks and initial blocks) will start executing at time 0 and will continue to execute concurrently. Sometimes this leads to race conditions, if coding is not done properly.

```
1  module multiple_blocks ();
2  reg a,b;
```

```

3 reg c,d;
4 reg clk,reset;
5 // Combo Logic
6 always @ ( c)
7 begin
8   a = c;
9 end
10 // Seq Logic
11 always @ (posedge clk)
12 if (reset) begin
13   b <= 0;
14 end else begin
15   b <= a & d;
16 end
17
18 // Testbench code here
19 initial begin
20   $monitor("TIME = %d CLK = %b C = %b D = %b A = %b B = %b",
21     $time, clk,c,d,a,b);
22   clk = 0;
23   reset = 0;
24   c = 0;
25   d = 0;
26   #2 reset = 1;
27   #2 reset = 0;
28   #2 c = 1;
29   #2 d = 1;
30   #2 c = 0;
31   #5 $finish;
32 end
33 // Clock generator
34 always
35   #1 clk = ~clk;
36
37 endmodule

```

You could download file multiple\_blocks.v [here](#)



## Race condition

```

1 module race_condition();
2 reg b;
3
4 initial begin
5   b = 0;
6 end
7

```

```

8 initial begin
9     b = 1;
10 end
11
12 endmodule

```

You could download file race\_condition.v [here](#)

In the code above it is difficult to say the value of b, as both blocks are supposed to execute at same time. In Verilog, if care is not taken, a race condition is something that occurs very often.



## Named Blocks

Blocks can be named by adding : block\_name after the keyword begin. Named blocks can be disabled using the 'disable' statement.



## Example - Named Blocks

```

1 // This code find the lowest bit set
2 module named_block_disable();
3
4 reg [31:0] bit_detect;
5 reg [5:0] bit_position;
6 integer i;
7
8 always @ (bit_detect)
9 begin : BIT_DETECT
10     for (i = 0; i < 32 ; i = i + 1) begin
11         // If bit is set, latch the bit position
12         // Disable the execution of the block
13         if (bit_detect[i] == 1) begin
14             bit_position = i;
15             disable BIT_DETECT;
16         end else begin
17             bit_position = 32;
18         end
19     end
20 end
21
22 // Testbench code here
23 initial begin
24     $monitor(" INPUT = %b MIN_POSITION = %d", bit_detect,
25 bit_position);
26     #1 bit_detect = 32'h1000_1000;
27     #1 bit_detect = 32'h1100_0000;
28     #1 bit_detect = 32'h1000_1010;

```

```

28   #10 $finish;
29 end
30
31 endmodule
  
```

You could download file named\_block\_disable.v [here](#)

In the example above, BIT\_DETECT is the named block and it is disabled whenever the bit position is detected.

### Procedural blocks and timing controls.

- Delay controls.
- Edge-Sensitive Event controls.
- Level-Sensitive Event controls-Wait statements.
- Named Events.



### Delay Controls

Delays the execution of a procedural statement by specific simulation time.

#< time > < statement >;



### Example - clk\_gen

```

1 module clk_gen ();
2
3 reg clk, reset;
4
5 initial begin
6   $monitor ("TIME = %g RESET = %b CLOCK = %b", $time,
reset, clk);
7   clk = 0;
8   reset = 0;
9   #2 reset = 1;
10  #5 reset = 0;
11  #10 $finish;
12 end
13
14 always
15   #1 clk = !clk;
16
  
```

### 17 endmodule

You could download file clk\_gen.v [here](#)

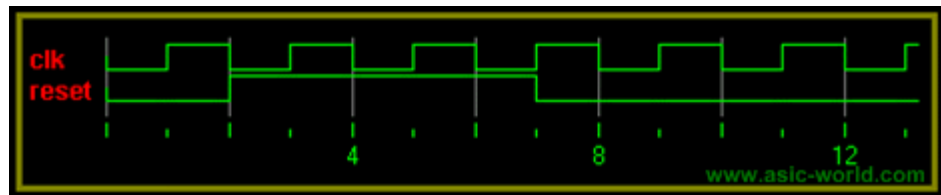
### Simulation Output

```

TIME = 0 RESET = 0 CLOCK = 0
TIME = 1 RESET = 0 CLOCK = 1
TIME = 2 RESET = 1 CLOCK = 0
TIME = 3 RESET = 1 CLOCK = 1
TIME = 4 RESET = 1 CLOCK = 0
TIME = 5 RESET = 1 CLOCK = 1
TIME = 6 RESET = 1 CLOCK = 0
TIME = 7 RESET = 0 CLOCK = 1
TIME = 8 RESET = 0 CLOCK = 0
TIME = 9 RESET = 0 CLOCK = 1
TIME = 10 RESET = 0 CLOCK = 0
TIME = 11 RESET = 0 CLOCK = 1
TIME = 12 RESET = 0 CLOCK = 0
TIME = 13 RESET = 0 CLOCK = 1
TIME = 14 RESET = 0 CLOCK = 0
TIME = 15 RESET = 0 CLOCK = 1
TIME = 16 RESET = 0 CLOCK = 0
  
```



### Waveform



### Edge sensitive Event Controls

Delays execution of the next statement until the specified transition on a signal.

**syntax :** @ (< posedge >|< negedge > signal) < statement >;





**Example - Edge Wait**

```
1 module edge_wait_example();
2
3 reg enable, clk, trigger;
4
5 always @ (posedge enable)
6 begin
7     trigger = 0;
8     // Wait for 5 clock cycles
9     repeat (5) begin
10         @ (posedge clk) ;
11     end
12     trigger = 1;
13 end
14
15 //Testbench code here
16 initial begin
17     $monitor ("TIME : %g CLK : %b ENABLE : %b TRIGGER : %b",
18         $time, clk,enable,trigger);
19     clk = 0;
20     enable = 0;
21     #5 enable = 1;
22     #1 enable = 0;
23     #10 enable = 1;
24     #1 enable = 0;
25     #10 $finish;
26 end
27
28 always
29     #1 clk = ~clk;
30
31 endmodule
```

You could download file edge\_wait\_example.v [here](#)

**Simulator Output**

```
TIME : 0 CLK : 0 ENABLE : 0 TRIGGER : x
TIME : 1 CLK : 1 ENABLE : 0 TRIGGER : x
TIME : 2 CLK : 0 ENABLE : 0 TRIGGER : x
TIME : 3 CLK : 1 ENABLE : 0 TRIGGER : x
TIME : 4 CLK : 0 ENABLE : 0 TRIGGER : x
TIME : 5 CLK : 1 ENABLE : 1 TRIGGER : 0
TIME : 6 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 7 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 8 CLK : 0 ENABLE : 0 TRIGGER : 0
```

```

TIME : 9 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 10 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 11 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 12 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 13 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 14 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 15 CLK : 1 ENABLE : 0 TRIGGER : 1
TIME : 16 CLK : 0 ENABLE : 1 TRIGGER : 0
TIME : 17 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 18 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 19 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 20 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 21 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 22 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 23 CLK : 1 ENABLE : 0 TRIGGER : 0
TIME : 24 CLK : 0 ENABLE : 0 TRIGGER : 0
TIME : 25 CLK : 1 ENABLE : 0 TRIGGER : 1
TIME : 26 CLK : 0 ENABLE : 0 TRIGGER : 1
  
```



### Level-Sensitive Even Controls ( Wait statements )

Delays execution of the next statement until < expression > evaluates to true

**syntax :** wait (< expression >) < statement >;



### Example - Level Wait

```

1 module wait_example();
2
3 reg mem_read, data_ready;
4 reg [7:0] data_bus, data;
5
6 always @ (mem_read or data_bus or data_ready)
7 begin
8   data = 0;
9   while (mem_read == 1'b1) begin
10     // #1 is very important to avoid infinite loop
11     wait (data_ready == 1) #1 data = data_bus;
12   end
13 end
14
15 // Testbench Code here
16 initial begin
17   $monitor ("TIME = %g READ = %b READY = %b DATA = %b",
18     $time, mem_read, data_ready, data);
19   data_bus = 0;
  
```

```
20 mem_read = 0;
21 data_ready = 0;
22 #10 data_bus = 8'hDE;
23 #10 mem_read = 1;
24 #20 data_ready = 1;
25 #1 mem_read = 1;
26 #1 data_ready = 0;
27 #10 data_bus = 8'hAD;
28 #10 mem_read = 1;
29 #20 data_ready = 1;
30 #1 mem_read = 1;
31 #1 data_ready = 0;
32 #10 $finish;
33 end
34
35 endmodule
```

You could download file wait\_example.v [here](#)

## Simulator Output

```
TIME = 0 READ = 0 READY = 0 DATA = 00000000
TIME = 20 READ = 1 READY = 0 DATA = 00000000
TIME = 40 READ = 1 READY = 1 DATA = 00000000
TIME = 41 READ = 1 READY = 1 DATA = 11011110
TIME = 42 READ = 1 READY = 0 DATA = 11011110
TIME = 82 READ = 1 READY = 1 DATA = 11011110
TIME = 83 READ = 1 READY = 1 DATA = 10101101
TIME = 84 READ = 1 READY = 0 DATA = 10101101
```



## Intra-Assignment Timing Controls

Intra-assignment controls always evaluate the right side expression immediately and assign the result after the delay or event control.

In non-intra-assignment controls (delay or event control on the left side), the right side expression is evaluated after the delay or event control.



## Example - Intra-Assignment

```
1 module intra_assign();
2
3 reg a, b;
4
5 initial begin
6     $monitor("TIME = %g A = %b B = %b", $time, a, b);
```

```

7    a = 1;
8    b = 0;
9    a = #10 0;
10   b = a;
11   #20 $display("TIME = %g A = %b B = %b", $time, a, b);
12   $finish;
13 end
14
15 endmodule

```

You could download file intra\_assign.v [here](#)

### Simulation Output

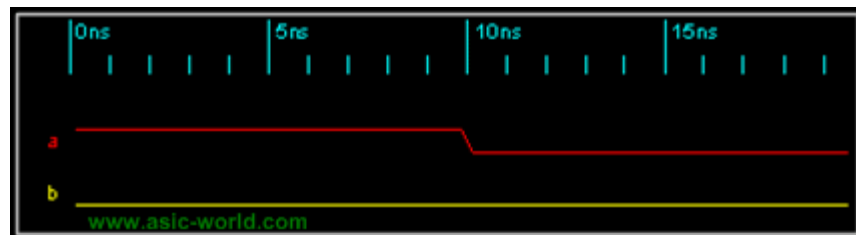
```

TIME = 0  A = 1 B = 0
TIME = 10 A = 0 B = 0
TIME = 30 A = 0 B = 0

```



### Waveform



### Modeling Combo Logic with Continuous Assignments

Whenever any signal changes on the right hand side, the entire right-hand side is re-evaluated and the result is assigned to the left hand side.



### Example - Tri-state Buffer

```

1 module tri_buf_using_assign();
2 reg data_in, enable;
3 wire pad;
4
5 assign pad = (enable) ? data_in : 1'bz;
6
7 initial begin
8   $monitor ("TIME = %g ENABLE = %b DATA : %b PAD %b",
9     $time, enable, data_in, pad);
10  #1 enable = 0;

```

```

11  #1 data_in = 1;
12  #1 enable = 1;
13  #1 data_in = 0;
14  #1 enable = 0;
15  #1 $finish;
16  end
17
18  endmodule

```

You could download file tri\_buf\_using\_assign.v [here](#)

### Simulation Output

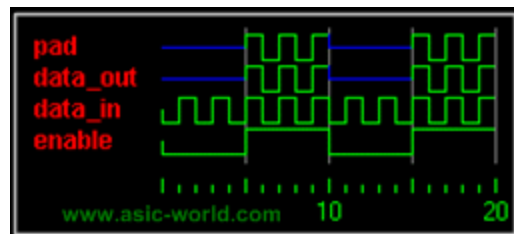
```

TIME = 0 ENABLE = x DATA : x PAD x
TIME = 1 ENABLE = 0 DATA : x PAD z
TIME = 2 ENABLE = 0 DATA : 1 PAD z
TIME = 3 ENABLE = 1 DATA : 1 PAD 1
TIME = 4 ENABLE = 1 DATA : 0 PAD 0
TIME = 5 ENABLE = 0 DATA : 0 PAD z

```



### Waveform



### Example - Mux

```

1  module mux_using_assign();
2  reg data_in_0, data_in_1;
3  wire data_out;
4  reg sel;
5
6  assign data_out = (sel) ? data_in_1 : data_in_0;
7
8  // Testbench code here
9  initial begin
10     $monitor("TIME = %g SEL = %b DATA0 = %b DATA1 = %b OUT = %b",
11             $time, sel, data_in_0, data_in_1, data_out);
12     data_in_0 = 0;
13     data_in_1 = 0;

```

```

14     sel = 0;
15     #10 sel = 1;
16     #10 $finish;
17 end
18
19 // Toggel data_in_0 at #1
20 always
21     #1 data_in_0 = ~data_in_0;
22
23 // Toggel data_in_1 at #2
24 always
25     #2 data_in_1 = ~data_in_1;
26
27 endmodule
  
```

You could download file mux\_using\_assign.v [here](#)

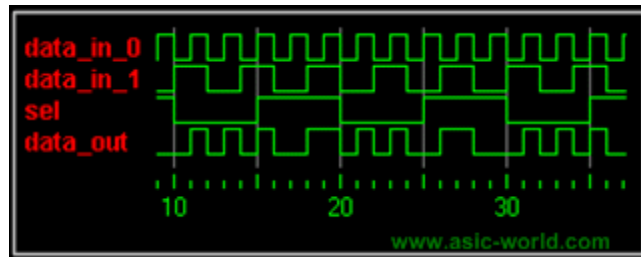
## Simulation Output

```

TIME = 0 SEL = 0 DATA0 = 0 DATA1 = 0 OUT = 0
TIME = 1 SEL = 0 DATA0 = 1 DATA1 = 0 OUT = 1
TIME = 2 SEL = 0 DATA0 = 0 DATA1 = 1 OUT = 0
TIME = 3 SEL = 0 DATA0 = 1 DATA1 = 1 OUT = 1
TIME = 4 SEL = 0 DATA0 = 0 DATA1 = 0 OUT = 0
TIME = 5 SEL = 0 DATA0 = 1 DATA1 = 0 OUT = 1
TIME = 6 SEL = 0 DATA0 = 0 DATA1 = 1 OUT = 0
TIME = 7 SEL = 0 DATA0 = 1 DATA1 = 1 OUT = 1
TIME = 8 SEL = 0 DATA0 = 0 DATA1 = 0 OUT = 0
TIME = 9 SEL = 0 DATA0 = 1 DATA1 = 0 OUT = 1
TIME = 10 SEL = 1 DATA0 = 0 DATA1 = 1 OUT = 1
TIME = 11 SEL = 1 DATA0 = 1 DATA1 = 1 OUT = 1
TIME = 12 SEL = 1 DATA0 = 0 DATA1 = 0 OUT = 0
TIME = 13 SEL = 1 DATA0 = 1 DATA1 = 0 OUT = 0
TIME = 14 SEL = 1 DATA0 = 0 DATA1 = 1 OUT = 1
TIME = 15 SEL = 1 DATA0 = 1 DATA1 = 1 OUT = 1
TIME = 16 SEL = 1 DATA0 = 0 DATA1 = 0 OUT = 0
TIME = 17 SEL = 1 DATA0 = 1 DATA1 = 0 OUT = 0
TIME = 18 SEL = 1 DATA0 = 0 DATA1 = 1 OUT = 1
TIME = 19 SEL = 1 DATA0 = 1 DATA1 = 1 OUT = 1
  
```



## Waveform



### Task

Tasks are used in all programming languages, generally known as procedures or subroutines. The lines of code are enclosed in task....end task brackets. Data is passed to the task, the processing done, and the result returned. They have to be specifically called, with data ins and outs, rather than just wired in to the general netlist. Included in the main body of code, they can be called many times, reducing code repetition.

- tasks are defined in the module in which they are used. It is possible to define a task in a separate file and use the compile directive 'include' to include the task in the file which instantiates the task.
- tasks can include timing delays, like posedge, negedge, # delay and wait.
- tasks can have any number of inputs and outputs.
- The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.
- tasks can take, drive and source global variables, when no local variables are used. When local variables are used, basically output is assigned only at the end of task execution.
- tasks can call another task or function.
- tasks can be used for modeling both combinational and sequential logic.
- A task must be specifically called with a statement, it cannot be used within an expression as a function can.



### Syntax

- A task begins with keyword task and ends with keyword endtask

- Inputs and outputs are declared after the keyword task.
- Local variables are declared after input and output declaration.



### Example - Simple Task

```
1 module simple_task();
2
3 task convert;
4 input [7:0] temp_in;
5 output [7:0] temp_out;
6 begin
7     temp_out = (9/5) * ( temp_in + 32)
8 end
9 endtask
10
11 endmodule
```

You could download file simple\_task.v [here](#)



### Example - Task using Global Variables

```
1 module task_global();
2
3 reg [7:0] temp_out;
4 reg [7:0] temp_in;
5
6 task convert;
7 begin
8     temp_out = (9/5) * ( temp_in + 32);
9 end
10 endtask
11
12 endmodule
```

You could download file task\_global.v [here](#)



### Calling a Task

Let's assume that the task in example 1 is stored in a file called mytask.v. Advantage of coding a task in a separate file, is that it can be used in multiple modules.

```
1 module task_calling (temp_a, temp_b, temp_c, temp_d);
```



```

2 input [7:0] temp_a, temp_c;
3 output [7:0] temp_b, temp_d;
4 reg [7:0] temp_b, temp_d;
5 `include "mytask.v"
6
7 always @ (temp_a)
8 begin
9   convert (temp_a, temp_b);
10 end
11
12 always @ (temp_c)
13 begin
14   convert (temp_c, temp_d);
15 end
16
17 endmodule

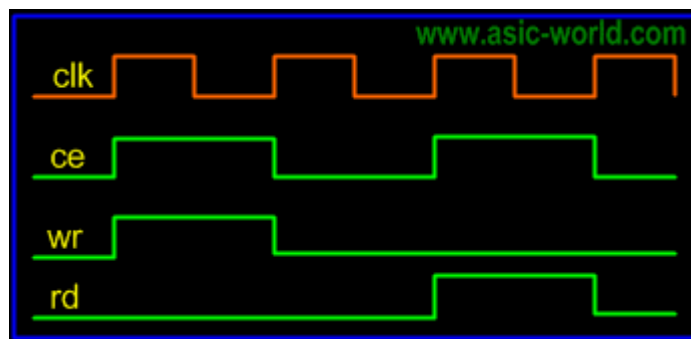
```

You could download file task\_calling.v [here](#)



### Example - CPU Write / Read Task

Below is the waveform used for writing into memory and reading from memory. We make the assumption that there is a need to use this interface from multiple agents. So we write the read/write as tasks.



```

1 module bus_wr_rd_task();
2
3 reg clk,rd,wr,ce;
4 reg [7:0] addr,data_wr,data_rd;
5 reg [7:0] read_data;
6
7 initial begin
8   clk = 0;
9   read_data = 0;
10  rd = 0;
11  wr = 0;
12  ce = 0;

```

```

13  addr = 0;
14  data_wr = 0;
15  data_rd = 0;
16  // Call the write and read tasks here
17  #1 cpu_write(8'h11,8'hAA);
18  #1 cpu_read(8'h11,read_data);
19  #1 cpu_write(8'h12,8'hAB);
20  #1 cpu_read(8'h12,read_data);
21  #1 cpu_write(8'h13,8'h0A);
22  #1 cpu_read(8'h13,read_data);
23  #100 $finish;
24  end
25  // Clock Generator
26  always
27    #1 clk = ~clk;
28  // CPU Read Task
29  task cpu_read;
30    input [7:0] address;
31    output [7:0] data;
32    begin
33      $display ("%g CPU Read task with address : %h", $time,
address);
34      $display ("%g -> Driving CE, RD and ADDRESS on to bus",
$time);
35      @ (posedge clk);
36      addr = address;
37      ce = 1;
38      rd = 1;
39      @ (negedge clk);
40      data = data_rd;
41      @ (posedge clk);
42      addr = 0;
43      ce = 0;
44      rd = 0;
45      $display ("%g CPU Read data : %h", $time, data);
46      $display ("=====");
47    end
48  endtask
49  // CU Write Task
50  task cpu_write;
51    input [7:0] address;
52    input [7:0] data;
53    begin
54      $display ("%g CPU Write task with address : %h Data : %h",
$time, address,data);
55      $display ("%g -> Driving CE, WR, WR data and ADDRESS on to
bus",
56      $time);
57

```

```

58      @ (posedge clk);
59      addr = address;
60      ce = 1;
61      wr = 1;
62      data_wr = data;
63      @ (posedge clk);
64      addr = 0;
65      ce = 0;
66      wr = 0;
67      $display ("=====");
68  end
69 endtask
70
71 // Memory model for checking tasks
72 reg [7:0] mem [0:255];
73
74 always @ (addr or ce or rd or wr or data_wr)
75 if (ce) begin
76   if (wr) begin
77     mem[addr] = data_wr;
78   end
79   if (rd) begin
80     data_rd = mem[addr];
81   end
82 end
83
84 endmodule

```

You could download file bus\_wr\_rd\_task.v [here](#)

## Simulation Output

```

1 CPU Write task with address : 11 Data : aa
1 -> Driving CE, WR, WR data and ADDRESS on to bus
=====
4 CPU Read task with address : 11
4 -> Driving CE, RD and ADDRESS on to bus
7 CPU Read data      : aa
=====
8 CPU Write task with address : 12 Data : ab
8 -> Driving CE, WR, WR data and ADDRESS on to bus
=====
12 CPU Read task with address : 12
12 -> Driving CE, RD and ADDRESS on to bus
15 CPU Read data      : ab
=====
16 CPU Write task with address : 13 Data : 0a
16 -> Driving CE, WR, WR data and ADDRESS on to bus
=====
20 CPU Read task with address : 13

```

```
20 -> Driving CE, RD and ADDRESS on to bus
23 CPU Read data      : 0a
=====
```



## Function

A Verilog HDL function is the same as a task, with very little differences, like function cannot drive more than one output, can not contain delays.

- functions are defined in the module in which they are used. It is possible to define functions in separate files and use compile directive 'include to include the function in the file which instantiates the task.
- functions **can not include timing delays**, like posedge, negedge, # delay, which means that functions should be executed in "zero" time delay.
- functions can have any number of inputs but only one output.
- The variables declared within the function are local to that function. The order of declaration within the function defines how the variables passed to the function by the caller are used.
- functions can take, drive, and source global variables, when no local variables are used. When local variables are used, basically output is assigned only at the end of function execution.
- functions can be used for **modeling combinational logic**.
- functions can **call other functions, but can not call tasks**.



## Syntax

- A function begins with keyword **function** and ends with keyword **endfunction**
- **inputs** are declared after the keyword function.



## Example - Simple Function

```
1 module simple_function();
2
3 function myfunction;
4 input a, b, c, d;
5 begin
```

```
6   myfunction = ((a+b) + (c-d)) ;  
7   end  
8   endfunction  
9  
10  endmodule
```

You could download file simple\_function.v [here](#)



### Example - Calling a Function

```
1  module  function_calling(a, b, c, d, e, f);  
2  
3  input  a, b, c, d, e ;  
4  output f;  
5  wire  f;  
6  `include "myfunction.v"  
7  
8  assign f = (myfunction (a,b,c,d)) ? e : 0;  
9  
10 endmodule
```

You could download file function\_calling.v [here](#)

## Introduction

There are tasks and functions that are used to generate input and output during simulation. Their names begin with a dollar sign (\$). The synthesis tools parse and ignore system functions, and hence can be included even in synthesizable models.



### \$display, \$strobe, \$monitor

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools like GTKWave. or Undertow or Debussy. \$display and \$strobe display once every time they are executed, whereas \$monitor displays every time one of its parameters changes. The difference between \$display and \$strobe is that \$strobe displays the parameters at the very end of the current simulation time unit rather than exactly when it is executed. The format string is like that in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time), %m (hierarchy level). %5d, %5b etc. would give exactly 5 spaces for the number

instead of the space needed. Append b, h, o to the task name to change default format to binary, octal or hexadecimal.



### Syntax

- \$display ("format\_string", par\_1, par\_2, ... );
- \$strobe ("format\_string", par\_1, par\_2, ... );
- \$monitor ("format\_string", par\_1, par\_2, ... );
- \$displayb (as above but defaults to binary..);
- \$strobeh (as above but defaults to hex..);
- \$monitoro (as above but defaults to octal..);



### \$time, \$stime, \$realtime

These return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively.



### \$reset, \$stop, \$finish

\$reset resets the simulation back to time 0; \$stop halts the simulator and puts it in interactive mode where the user can enter commands; \$finish exits the simulator back to the operating system.



### \$scope, \$showscope

\$scope(hierarchy\_name) sets the current hierarchical scope to hierarchy\_name. \$showscopes(n) lists all modules, tasks and block names in (and below, if n is set to 1) the current scope.



### \$random

\$random generates a random integer every time it is called. If the sequence is to be repeatable, the first time one invokes random giving it a numerical argument (a seed). Otherwise the seed is derived from the computer clock.



### \$dumpfile, \$dumpvar, \$dumpon, \$dumpoff, \$dumpall

These can dump variable changes to a simulation viewer like Debussy. The dump files are capable of dumping all the variables in a simulation. This is convenient for debugging, but can be very slow.



### Syntax

- `$dumpfile("filename.vcd")`
- `$dumpvar` dumps all variables in the design.
- `$dumpvar(1, top)` dumps all the variables in module top and below, but not modules instantiated in top.
- `$dumpvar(2, top)` dumps all the variables in module top and 1 level below.
- `$dumpvar(n, top)` dumps all the variables in module top and n-1 levels below.
- `$dumpvar(0, top)` dumps all the variables in module top and all level below.
- `$dumpon` initiates the dump.
- `$dumpoff` stop dumping.



### **`$fopen`, `$fdisplay`, `$fstrobe` `$fmonitor` and `$fwrite`**

These commands write more selectively to files.

- `$fopen` opens an output file and gives the open file a handle for use by the other commands.
- `$fclose` closes the file and lets other programs access it.
- `$fdisplay` and `$fwrite` write formatted data to a file whenever they are executed. They are the same except `$fdisplay` inserts a new line after every execution and `$fwrite` does not.
- `$fstrobe` also writes to a file when executed, but it waits until all other operations in the time step are complete before writing. Thus initial #1 a=1; b=0; `$fstrobe(hand1, a,b); b=1;` will write 1 1 for a and b.
- `$fmonitor` writes to a file whenever any of its arguments changes.



### **Syntax**

- `handle1=$fopen("filenam1.suffix")`
- `handle2=$fopen("filenam2.suffix")`
- `$fstrobe(handle1, format, variable list) //strobe data into filenam1.suffix`
- `$fdisplay(handle2, format, variable list) //write data into filenam2.suffix`

- `$fwrite(handle2, format, variable list) //write data into`  
`filenam2.suffix` all on one line. Put in the format string where  
 a new line is desired.

Writing a testbench is as complex as writing the RTL code its. These days ASICs are getting more and more complex and the verifying these complex ASIC has become a challenge. Typically 60-70% of time needed for any ASIC is spent on verification/validation/testing. Even though the above facts are well known to most ASIC engineers, still engineers think that there is no glory in verification.

I have picked up some examples from the VLSI classes that I used to teach during 1999-2001, when I was in Chennai. Please feel free to give your feedback on how to improve the tutorials below.



### Before you Start

For writing testbenches it is important to have the design specification of "design under test" or simply DUT. Specs need to be understood clearly and a test plan, which basically documents the test bench architecture and the test scenarios (test cases) in detail, needs to be made.



### Example - Counter

Let's assume that we have to verify a simple 4-bit up counter which increments its count whenever enable is high, and resets to zero when reset is asserted high. Reset is synchronous to clock.



### Code for Counter

```

1 //-----
2 // Design Name : counter
3 // File Name   : counter.v
4 // Function    : 4 bit up counter
5 // Coder       : Deepak
6 //-----
7 module counter (clk, reset, enable, count);
8 input clk, reset, enable;
```



```

9  output [3:0] count;
10 reg [3:0] count;
11
12 always @ (posedge clk)
13 if (reset == 1'b1) begin
14   count <= 0;
15 end else if (enable == 1'b1) begin
16   count <= count + 1;
17 end
18
19 endmodule

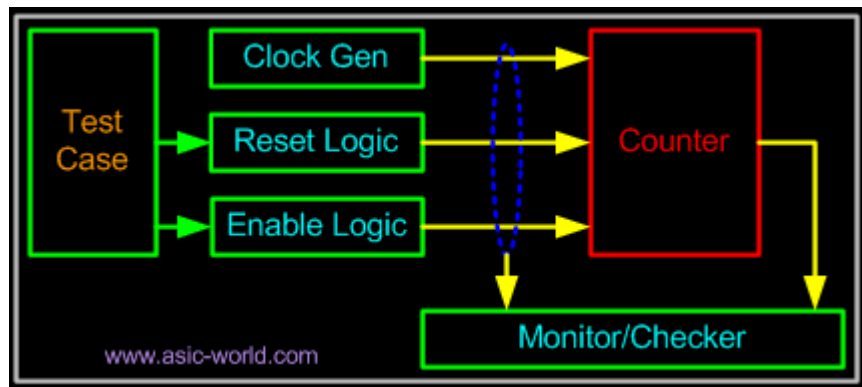
```

You could download file counter.v [here](#)



### Test Plan

We will write a self-checking test bench, but we will do this in steps to help you understand the concept of writing automated test benches. Our testbench environment will look something like the figure below.



DUT is instantiated in the testbench, and the testbench will contain a clock generator, reset generator, enable logic generator and compare logic, which basically calculates the expected count value of counter and compares it with the output of counter.



### Test Cases

- **Reset Test :** We can start with reset de-asserted, followed by asserting reset for few clock ticks and deasserting the reset, See if counter sets its output to zero.

- Enable Test : Assert/deassert enable after reset is applied.
- Random Assert/deassert of enable and reset.

We can add some more test cases; but we are not here to test the counter, rather to learn how to write test benches.

### Writing a TestBench

First step of any testbench creation is building a dummy template which basically declares inputs to DUT as reg and outputs from DUT as wire, then instantiates the DUT as shown in the code below. Note that there is no port list for the test bench.



#### Test Bench

```
1 module counter_tb;
2   reg clk, reset, enable;
3   wire [3:0] count;
4
5   counter U0 (
6     .clk      (clk),
7     .reset    (reset),
8     .enable   (enable),
9     .count    (count)
10  );
11
12 endmodule
```

You could download file counter\_tb1.v [here](#)

Next step would be to add clock generator logic: this is straight forward, as we know how to generate a clock. Before we add a clock generator we need to drive all the inputs to DUT to some known state as shown in the code below.



#### Test Bench with Clock generator

```
1 module counter_tb;
2   reg clk, reset, enable;
```

```
3  wire [3:0] count;
4
5  counter U0 (
6    .clk      (clk) ,
7    .reset    (reset) ,
8    .enable   (enable) ,
9    .count    (count)
10 );
11
12 initial
13 begin
14     clk = 0;
15     reset = 0;
16     enable = 0;
17 end
18
19 always
20     #5 clk = !clk;
21
22 endmodule
```

You could download file counter\_tb2.v [here](#)

An initial block in Verilog is executed only once, thus simulator sets the value of clk, reset and enable to 0; by looking at the counter code (of course you will be referring to the DUT specs) could be found that driving 0 makes all these signals disabled.

There are many ways to generate a clock: one could use a forever loop inside an initial block as an alternative to the above code. You could add a parameter or use `define to control the clock frequency. You may write a complex clock generator, where we could introduce PPM (Parts per million, clock width drift), then control the duty cycle. All the above depends on the specs of the DUT and the creativity of a "Test Bench Designer".

At this point, you would like to test if the testbench is generating the clock correctly: well you can compile it with any Verilog simulator. You need to give command line options as shown below.

```
C:\www.asic-world.com\veridos counter.v counter_tb.v
```

Of course it is a very good idea to keep file names the same as the module name. Ok, coming back to compiling, you will see that the simulator does print anything on screen, or dump any waveform. Thus we need to add support for all the above as shown in the code below.

**Test Bench continues...**

```
1 module counter_tb;
2   reg clk, reset, enable;
3   wire [3:0] count;
4
5   counter U0 (
6     .clk      (clk),
7     .reset    (reset),
8     .enable   (enable),
9     .count    (count)
10  );
11
12  initial begin
13    clk = 0;
14    reset = 0;
15    enable = 0;
16  end
17
18  always
19    #5 clk = !clk;
20
21  initial begin
22    $dumpfile ("counter.vcd");
23    $dumpvars;
24  end
25
26  initial begin
27    $display ("%t\ttime,%tclk,%treset,%tenable,%tcount");
28    $monitor ("%d,%tb,%tb,%tb,%td", $time,
29      clk, reset, enable, count);
30  end
31
32  initial
33    #100 $finish;
34
35    //Rest of testbench code after this line
36  endmodule
```

You could download file counter\_tb3.v [here](#)

\$dumpfile is used for specifying the file that the simulator will use to store the waveform, that can be used later using a waveform viewer. (Please refer to the tools section for freeware versions of viewers.) \$dumpvars basically instructs the Verilog compiler to start dumping all the signals to "counter.vcd".

\$display is used for printing text or variables to stdout (screen), \t is for inserting tabs. The syntax is the same as for printf C language. \$monitor in the second line is a bit different: \$monitor keeps track of changes to the variables that are in the list (clk, reset, enable, count). Whenever any of them changes, it prints their value, in the respective radix specified.

\$finish is used for terminating the simulation after #100 time units (note: all the initial, always blocks start execution at time 0).

Now that we have written the basic skeleton, let's compile and see what we have just coded. Output of the simulator is shown below.

```
C:\www.asic-world.com>veridos counter.v counter_tb.v
VeriWell for Win32 HDL Version 2.1.4 Fri Jan 17 21:33:25 2003

This is a free version of the VeriWell for Win32 Simulator
Distribute this freely; call 1-800-VERIWELL for ordering information
See the file "!readme.1st" for more information

Copyright (c) 1993-97 Wellspring Solutions, Inc.
All rights reserved

Memory Available: 0
Entering Phase I...
Compiling source file : counter.v
Compiling source file : counter_tb.v
The size of this model is [2%, 5%] of the capacity of the free version

Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
counter_tb

    time  clk,      reset,   enable, count
    0,      0,        0,        0,        0,        x
    5,      1,        1,        0,        0,        x
    10,     0,        0,        0,        x
    15,     1,        0,        0,        x
    20,     0,        0,        0,        x
    25,     1,        0,        0,        x
    30,     0,        0,        0,        x
    35,     1,        0,        0,        x
    40,     0,        0,        0,        x
    45,     1,        0,        0,        x
    50,     0,        0,        0,        x
```

```

55,  1,    0,    0,    x
60,  0,    0,    0,    x
65,  1,    0,    0,    x
70,  0,    0,    0,    x
75,  1,    0,    0,    x
80,  0,    0,    0,    x
85,  1,    0,    0,    x
90,  0,    0,    0,    x
95,  1,    0,    0,    x
  
```

```

Exiting VeriWell for Win32 at time 100
0 Errors, 0 Warnings, Memory Used: 0
Compile time = 0.0 Load time = 0.0 Simulation time = 0.1
  
```

```

Normal exit
Thank you for using VeriWell for Win32
  
```

## Adding Reset Logic

Once we have the basic logic to allow us to see what our testbench is doing, we can next add the reset logic. If we look at the testcases, we see that we had added a constraint that it should be possible to activate reset anytime during simulation. To achieve this we have many approaches, but I am going to teach something that will go long way. There is something called 'events' in Verilog: events can be triggered, and also monitored, to see if an event has occurred.

Let's code our reset logic in such a way that it waits for the trigger event "reset\_trigger": when this event happens, reset logic asserts reset at negative edge of clock and de-asserts on next negative edge as shown in the code below. Also after de-asserting the reset, reset logic triggers another event called "reset\_done\_trigger". This trigger event can then be used somewhere else in the testbench to sync up.

### Code of reset logic

```

1 event reset_trigger;
2 event reset_done_trigger;
3
4 initial begin
5   forever begin
6     @ (reset_trigger);
7     @ (negedge clk);
8     reset = 1;
  
```

```

9      @ (negedge clk);
10     reset = 0;
11     -> reset_done_trigger;
12 end
13 end

```

You could download file counter\_tb4.v [here](#)

### ✦ Adding test case logic

Moving forward, let's add logic to generate the test cases, ok we have three testcases as in the first part of this tutorial. Let's list them again. 😊

- Reset Test : We can start with reset de-asserted, followed by asserting reset for few clock ticks and de-asserting the reset, See if counter sets its output to zero.
- Enable Test : Assert/de-assert enable after reset is applied.
- Random Assert/de-assert of enable and reset.

Repeating it again: "There are many ways" to code a test case, it all depends on the creativity of the Test bench designer. Let's take a simple approach and then slowly build upon it.

### ✦ Test Case 1 - Asserting/ De-asserting reset

In this test case, we will just trigger the event reset\_trigger after 10 simulation units.

```

1 initial
2   begin: TEST_CASE
3     #10 -> reset_trigger;
4   end

```

You could download file counter\_tb5.v [here](#)

### ✦ Test Case 2 - Assert/ De-assert enable after reset is applied.

In this test case, we will trigger the reset logic and wait for the reset logic to complete its operation, before we start driving the enable signal to logic 1.

```

1 initial
2   begin: TEST_CASE
3     #10 -> reset_trigger;

```

```

4      @ (reset_done_trigger);
5      @ (negedge clk);
6      enable = 1;
7      repeat (10) begin
8          @ (negedge clk);
9      end
10     enable = 0;
11 end

```

You could download file counter\_tb6.v [here](#)

### ✦ Test Case 3 - Assert/De-assert enable and reset randomly.

In this testcase we assert the reset, and then randomly drive values on to enable and reset signal.

```

1  initial
2  begin : TEST_CASE
3      #10 -> reset_trigger;
4      @ (reset_done_trigger);
5      fork
6          repeat (10) begin
7              @ (negedge clk);
8              enable = $random;
9          end
10         repeat (10) begin
11             @ (negedge clk);
12             reset = $random;
13         end
14     join
15 end

```

You could download file counter\_tb7.v [here](#)

Well you might ask, do all this three test case exist in same file? Well, the answer is no. If we try to have all three test cases on one file, then we end up having race conditions due to three initial blocks driving reset and enable signal. So normally, once test bench coding is done, test cases are coded separately and included in testbench with `include directives as shown below. (There are better ways to do this, but you have to think how you want to do it).

If you look closely at all the three test cases, you will find that even though test case execution is not complete, simulation terminates. To have better control, what we can do is adding an event like "terminate\_sim" and execute \$finish only when this event is triggered. We can trigger this event at the end of test case execution. The code for \$finish now could look as shown below.



```
1 event terminate_sim;  
2 initial begin  
3   @ (terminate_sim);  
4   #5 $finish;  
5 end
```

You could download file counter\_tb8.v [here](#)

The modified test case #2 would be like:

```
1 initial  
2   begin: TEST_CASE  
3     #10 -> reset_trigger;  
4     @ (reset_done_trigger);  
5     @ (negedge clk);  
6     enable = 1;  
7     repeat (10) begin  
8       @ (negedge clk);  
9     end  
10    enable = 0;  
11    #5 -> terminate_sim;  
12  end  
13
```

You could download file counter\_tb9.v [here](#)

Second problem with the approach that we have taken till now is that we need to manually check the waveform and also the simulator output on the screen to see if the DUT is working correctly. Part IV shows how to automate this.

### Adding compare Logic

To make any testbench self checking/automated, first we need to develop a model that mimics the DUT in functionality. In our example, it's going to be very easy, but at times if the DUT is complex, then to mimic it will be very complex and will require a lot of innovative techniques to make self-checking work.

```
1 reg [3:0] count_compare;  
2  
3 always @ (posedge clk)  
4 if (reset == 1'b1) begin
```

```
5    count_compare <= 0;
6 end else if ( enable == 1'b1) begin
7    count_compare <= count_compare + 1;
8 end
```

You could download file counter\_tb10.v [here](#)

Once we have the logic to mimic the DUT functionality, we need to add the checker logic, which at any given point keeps checking the expected value with the actual value. Whenever there is any error, it prints out the expected and actual value, and also terminates the simulation by triggering the event "terminate\_sim".

```
1 always @ (posedge clk)
2   if (count_compare != count) begin
3     $display ("DUT Error at time %d", $time);
4     $display (" Expected value %d, Got Value %d", count_compare,
count);
5     #5 -> terminate_sim;
6   end
```

You could download file counter\_tb11.v [here](#)

Now that we have the all the logic in place, we can remove \$display and \$monitor, as our testbench have become fully automatic, and we don't require to manually verify the DUT input and output. Try changing the count\_compare = count\_compare + 2, and see how compare logic works. This is just another way to see if our testbench is stable.

We could add some fancy printing as shown in the figure below to make our test environment more friendly.

```
C:\Download\work>veridos counter.v counter_tb.v
VeriWell for Win32 HDL  Sat Jan 18 20:10:35 2003
```

This is a free version of the VeriWell for Win32 Simulator  
Distribute this freely; call 1-800-VERIWELL for ordering information  
See the file "!readme.1st" for more information

Copyright (c) 1993-97 Wellspring Solutions, Inc.  
All rights reserved

Memory Available: 0  
Entering Phase I...  
Compiling source file : counter.v  
Compiling source file : counter\_tb.v

The size of this model is [5%, 6%] of the capacity of the free version

```
Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
counter_tb

#####
Applying reset
Came out of Reset
Terminating simulation
Simulation Result : PASSED
#####
Exiting VeriWell for Win32 at time 96
0 Errors, 0 Warnings, Memory Used: 0
Compile time = 0.0, Load time = 0.0, Simulation time = 0.0

Normal exit
Thank you for using VeriWell for Win32
```

I know, you would like to see the test bench code that I used to generate the above output, well you can find it [here](#) and counter code [here](#).

There are a lot of things that I have not covered; maybe when I find time, I may add some more details on this subject.

As to books, I am yet to find a good book on writing test benches.

## Memory Modeling

To help modeling of memory, Verilog provides support for two dimensions arrays. Behavioral models of memories are modeled by declaring an array of register variables; any word in the array may be accessed using an index into the array. A temporary variable is required to access a discrete bit within the array.



### Syntax

```
reg [wordsize:0] array_name [0:arraysize]
```



### Examples



### Declaration

```
reg [7:0] my_memory [0:255];
```

Here [7:0] is the memory width and [0:255] is the memory depth with the following parameters:

- Width : 8 bits, little endian
- Depth : 256, address 0 corresponds to location 0 in the array.

#### ✦ Storing Values

```
my_memory[address] = data_in;
```

#### ✦ Reading Values

```
data_out = my_memory[address];
```

#### ✦ Bit Read

Sometimes there may be need to read just one bit. Unfortunately Verilog does not allow to read or write only one bit: the workaround for such a problem is as shown below.

```
data_out = my_memory[address];
```

```
data_out_it_0 = data_out[0];
```

#### ❖ Initializing Memories

A memory array may be initialized by reading memory pattern file from disk and storing it on the memory array. To do this, we use system tasks \$readmemb and \$readmemh. \$readmemb is used for binary representation of memory content and \$readmemh for hex representation.

#### ✦ Syntax

```
$readmemh("file_name",mem_array,start_addr,stop_addr);
```

Note : start\_addr and stop\_addr are optional.

#### ✦ Example - Simple memory

```
1 module memory();  
2 reg [7:0] my_memory [0:255];  
3  
4 initial begin  
5     $readmemh("memory.list", my_memory);
```

```
6 end
7 endmodule
```

You could download file memory.v [here](#)

### ✦ Example - Memory.list file

```
1 //Comments are allowed
2 1100_1100 // This is first address i.e 8'h00
3 1010_1010 // This is second address i.e 8'h01
4 @ 55 // Jump to new address 8'h55
5 0101_1010 // This is address 8'h55
6 0110_1001 // This is address 8'h56
```

You could download file memory.list [here](#)

\$readmemh system task can also be used for reading testbench vectors. I will cover this in detail in the test bench section ... when I find time.

Refer to the examples section for more details on different types of memories.

## Introduction to FSM

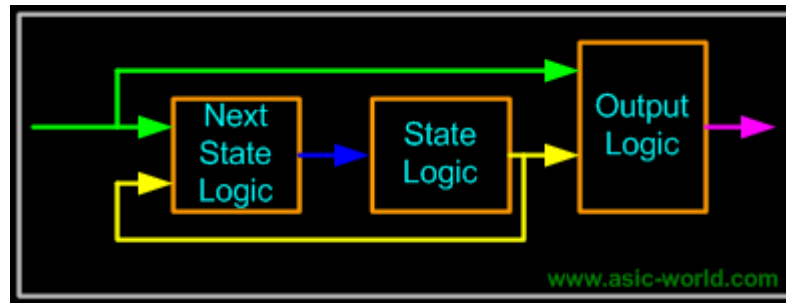
State machines or FSM are the heart of any digital design; of course a counter is a simple form of FSM. When I was learning Verilog, I used to wonder "How do I code FSM in Verilog" and "What is the best way to code it". I will try to answer the first part of the question below and second part of the question can be found in the tidbits section.

### State machine Types

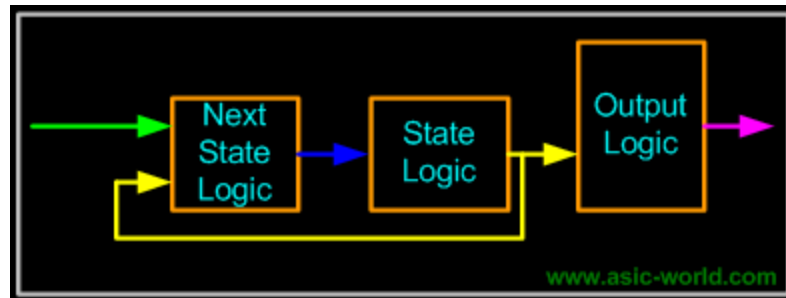
There are two types of state machines as classified by the types of outputs generated from each. The first is the Moore State Machine where the outputs are only a function of the present state, the second is the Mealy State Machine where one or more of the outputs are a function of the present state and one or more of the inputs.



### Mealy Model



### Moore Model



State machines can also be classified according to the state encoding used. Encoding style is also a critical factor which decides speed and gate complexity of the FSM. Binary, gray, one hot, one cold, and almost one hot are the different types of encoding styles used in coding FSM states.



### Modeling State machines.

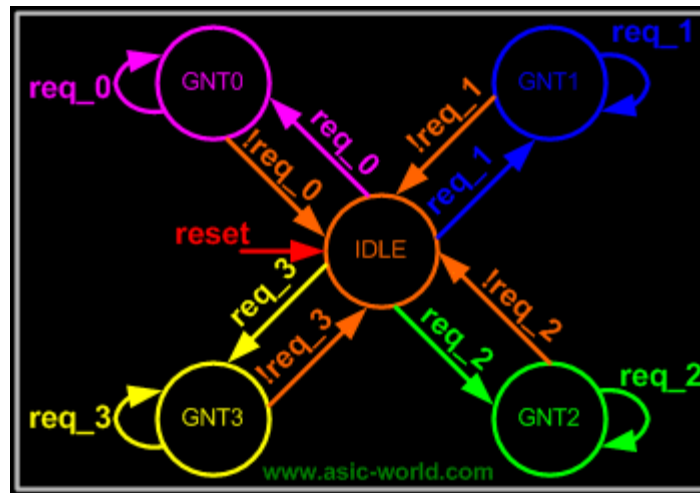
One thing that need to be kept in mind when coding FSM is that combinational logic and sequence logic should be in two different always blocks. In the above two figures, next state logic is always the combinational logic. State Registers and Output logic are sequential logic. It is very important that any asynchronous signal to the next state logic be synchronized before being fed to the FSM. Always try to keep FSM in a separate Verilog file.

Using constants declaration like parameter or `define to define states of the FSM makes code more readable and easy to manage.



### Example - Arbiter

We will be using the arbiter FSM to study FSM coding styles in Verilog.



### Verilog Code

FSM code should have three sections:

- Encoding style.
- Combinational part.
- Sequential part.



### Encoding Style

There are many encoding styles around, some of which are:

- Binary Encoding
- One Hot Encoding
- One Cold Encoding
- Almost One Hot Encoding
- Almost One Cold Encoding
- Gray Encoding

Of all the above types we normally use one hot and binary encoding.



### One Hot Encoding

```

1 parameter [4:0] IDLE = 5'b0_0001;
2 parameter [4:0] GNT0 = 5'b0_0010;
3 parameter [4:0] GNT1 = 5'b0_0100;
4 parameter [4:0] GNT2 = 5'b0_1000;
5 parameter [4:0] GNT3 = 5'b1_0000;
  
```

You could download file fsm\_one\_hot\_params.v [here](#)



## Binary Encoding

```

1 parameter [2:0] IDLE = 3'b000;
2 parameter [2:0] GNT0 = 3'b001;
3 parameter [2:0] GNT1 = 3'b010;
4 parameter [2:0] GNT2 = 3'b011;
5 parameter [2:0] GNT3 = 3'b100;

```

You could download file fsm\_binary\_params.v [here](#)



## Gray Encoding

```

1 parameter [2:0] IDLE = 3'b000;
2 parameter [2:0] GNT0 = 3'b001;
3 parameter [2:0] GNT1 = 3'b011;
4 parameter [2:0] GNT2 = 3'b010;
5 parameter [2:0] GNT3 = 3'b110;

```

You could download file fsm\_gray\_params.v [here](#)

## Combinational Section

This section can be modeled using functions, assign statements or using always blocks with a case statement. For the time being let's see the always block version

```

1 always @ (state or req_0 or req_1 or req_2 or req_3)
2 begin
3   next_state = 0;
4   case (state)
5     IDLE : if (req_0 == 1'b1) begin
6       next_state = GNT0;
7     end else if (req_1 == 1'b1) begin
8       next_state = GNT1;
9     end else if (req_2 == 1'b1) begin
10      next_state = GNT2;
11    end else if (req_3 == 1'b1) begin
12      next_state = GNT3;
13    end else begin
14      next_state = IDLE;
15    end
16   GNT0 : if (req_0 == 1'b0) begin
17     next_state = IDLE;

```



```

18         end else begin
19             next_state = GNT0;
20         end
21     GNT1 : if (req_1 == 1'b0) begin
22         next_state = IDLE;
23         end else begin
24             next_state = GNT1;
25         end
26     GNT2 : if (req_2 == 1'b0) begin
27         next_state = IDLE;
28         end else begin
29             next_state = GNT2;
30         end
31     GNT3 : if (req_3 == 1'b0) begin
32         next_state = IDLE;
33         end else begin
34             next_state = GNT3;
35         end
36     default : next_state = IDLE;
37 endcase
38 end
  
```

You could download file fsm\_combo.v [here](#)

### Sequential Section

This section has to be modeled using only edge sensitive logic such as always block with posedge or negedge of clock.

```

1 always @ (posedge clock)
2 begin : OUTPUT_LOGIC
3     if (reset == 1'b1) begin
4         gnt_0 <= #1 1'b0;
5         gnt_1 <= #1 1'b0;
6         gnt_2 <= #1 1'b0;
7         gnt_3 <= #1 1'b0;
8         state <= #1 IDLE;
9     end else begin
10        state <= #1 next_state;
11        case (state)
12            IDLE : begin
13                gnt_0 <= #1 1'b0;
14                gnt_1 <= #1 1'b0;
15                gnt_2 <= #1 1'b0;
16                gnt_3 <= #1 1'b0;
17            end
18        GNT0 : begin
  
```

```

19         gnt_0 <= #1 1'b1;
20     end
21     GNT1 : begin
22         gnt_1 <= #1 1'b1;
23     end
24     GNT2 : begin
25         gnt_2 <= #1 1'b1;
26     end
27     GNT3 : begin
28         gnt_3 <= #1 1'b1;
29     end
30     default : begin
31         state <= #1 IDLE;
32     end
33 endcase
34 end
35 end
  
```

You could download file fsm\_seq.v [here](#)

### Full Code using binary encoding

```

1  module fsm_full(
2  clock , // Clock
3  reset , // Active high reset
4  req_0 , // Active high request from agent 0
5  req_1 , // Active high request from agent 1
6  req_2 , // Active high request from agent 2
7  req_3 , // Active high request from agent 3
8  gnt_0 , // Active high grant to agent 0
9  gnt_1 , // Active high grant to agent 1
10 gnt_2 , // Active high grant to agent 2
11 gnt_3 , // Active high grant to agent 3
12 );
13 // Port declaration here
14 input clock ; // Clock
15 input reset ; // Active high reset
16 input req_0 ; // Active high request from agent 0
17 input req_1 ; // Active high request from agent 1
18 input req_2 ; // Active high request from agent 2
19 input req_3 ; // Active high request from agent 3
20 output gnt_0 ; // Active high grant to agent 0
21 output gnt_1 ; // Active high grant to agent 1
22 output gnt_2 ; // Active high grant to agent 2
23 output gnt_3 ; // Active high grant to agent
24
25 // Internal Variables
  
```

```

26 reg    gnt_0 ; // Active high grant to agent 0
27 reg    gnt_1 ; // Active high grant to agent 1
28 reg    gnt_2 ; // Active high grant to agent 2
29 reg    gnt_3 ; // Active high grant to agent
30
31 parameter [2:0] IDLE  = 3'b000;
32 parameter [2:0] GNT0  = 3'b001;
33 parameter [2:0] GNT1  = 3'b010;
34 parameter [2:0] GNT2  = 3'b011;
35 parameter [2:0] GNT3  = 3'b100;
36
37 reg [2:0] state, next_state;
38
39 always @ (state or req_0 or req_1 or req_2 or req_3)
40 begin
41   next_state = 0;
42   case(state)
43     IDLE : if (req_0 == 1'b1) begin
44       next_state = GNT0;
45     end else if (req_1 == 1'b1) begin
46       next_state = GNT1;
47     end else if (req_2 == 1'b1) begin
48       next_state = GNT2;
49     end else if (req_3 == 1'b1) begin
50       next_state = GNT3;
51     end else begin
52       next_state = IDLE;
53     end
54     GNT0 : if (req_0 == 1'b0) begin
55       next_state = IDLE;
56     end else begin
57       next_state = GNT0;
58     end
59     GNT1 : if (req_1 == 1'b0) begin
60       next_state = IDLE;
61     end else begin
62       next_state = GNT1;
63     end
64     GNT2 : if (req_2 == 1'b0) begin
65       next_state = IDLE;
66     end else begin
67       next_state = GNT2;
68     end
69     GNT3 : if (req_3 == 1'b0) begin
70       next_state = IDLE;
71     end else begin
72       next_state = GNT3;
73   end

```

```

74     default : next_state = IDLE;
75   endcase
76 end
77
78 always @ (posedge clock)
79 begin : OUTPUT_LOGIC
80   if (reset) begin
81     gnt_0 <= #1 1'b0;
82     gnt_1 <= #1 1'b0;
83     gnt_2 <= #1 1'b0;
84     gnt_3 <= #1 1'b0;
85     state <= #1 IDLE;
86   end else begin
87     state <= #1 next_state;
88     case (state)
89     IDLE : begin
90         gnt_0 <= #1 1'b0;
91         gnt_1 <= #1 1'b0;
92         gnt_2 <= #1 1'b0;
93         gnt_3 <= #1 1'b0;
94       end
95     GNT0 : begin
96         gnt_0 <= #1 1'b1;
97       end
98     GNT1 : begin
99         gnt_1 <= #1 1'b1;
100     end
101     GNT2 : begin
102         gnt_2 <= #1 1'b1;
103     end
104     GNT3 : begin
105         gnt_3 <= #1 1'b1;
106     end
107     default : begin
108         state <= #1 IDLE;
109     end
110   endcase
111 end
112 end
113
114 endmodule

```

You could download file fsm\_full.v [here](#)

## Testbench

```

1  `include "fsm_full.v"
2
3  module fsm_full_tb();

```

```

4 reg clock , reset ;
5 reg req_0 , req_1 , req_2 , req_3;
6 wire gnt_0 , gnt_1 , gnt_2 , gnt_3 ;
7
8 initial begin
9     $display("Time\t R0 R1 R2 R3 G0 G1 G2 G3");
10    $monitor("%g\t %b %b %b %b %b %b %b",
11        $time, req_0, req_1, req_2, req_3, gnt_0, gnt_1,
gnt_2, gnt_3);
12    clock = 0;
13    reset = 0;
14    req_0 = 0;
15    req_1 = 0;
16    req_2 = 0;
17    req_3 = 0;
18    #10 reset = 1;
19    #10 reset = 0;
20    #10 req_0 = 1;
21    #20 req_0 = 0;
22    #10 req_1 = 1;
23    #20 req_1 = 0;
24    #10 req_2 = 1;
25    #20 req_2 = 0;
26    #10 req_3 = 1;
27    #20 req_3 = 0;
28    #10 $finish;
29 end
30
31 always
32     #2 clock = ~clock;
33
34
35 fsm_full U_fsm_full(
36 clock , // Clock
37 reset , // Active high reset
38 req_0 , // Active high request from agent 0
39 req_1 , // Active high request from agent 1
40 req_2 , // Active high request from agent 2
41 req_3 , // Active high request from agent 3
42 gnt_0 , // Active high grant to agent 0
43 gnt_1 , // Active high grant to agent 1
44 gnt_2 , // Active high grant to agent 2
45 gnt_3 // Active high grant to agent 3
46 );
47
48
49
50 endmodule

```

You could download file fsm\_full\_tb.v [here](#)

### Simulator Output

| Time | R0 | R1 | R2 | R3 | G0 | G1 | G2 | G3 |
|------|----|----|----|----|----|----|----|----|
| 0    | 0  | 0  | 0  | 0  | x  | x  | x  | x  |
| 7    | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 30   | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 35   | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |
| 50   | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |
| 55   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 60   | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |
| 67   | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  |
| 80   | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |
| 87   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 90   | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |
| 95   | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  |
| 110  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
| 115  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 120  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |
| 127  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  |
| 140  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |
| 147  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

## Introduction

Let's assume that we have a design which requires us to have counters of various width, but with the same functionality. Maybe we can assume that we have a design which requires lots of instances of different depth and width of RAMs of similar functionality. Normally what we do is creating counters of different widths and then use them. The same rule applies to the RAM we talked about.

But Verilog provides a powerful way to overcome this problem: it provides us with something called parameter; these parameters are like constants local to that particular module.

We can override the default values, either using defparam or by passing a new set of parameters during instantiation. We call this parameter overriding.



## Parameters

A parameter is defined by Verilog as a constant value declared within the module structure. The value can be used to define a set of attributes for the module which can characterize its behavior as well as its physical representation.

- Defined inside a module.
- Local scope.
- Maybe overridden at instantiation time.
  - If multiple parameters are defined, they must be overridden in the order they were defined. If an overriding value is not specified, the default parameter declaration values are used.
- Maybe changed using the defparam statement.



### Parameter Override using defparam

```
1 module secret_number;
2 parameter my_secret = 0;
3
4 initial begin
5     $display("My secret number is %d", my_secret);
6 end
7
8 endmodule
9
10 module defparam_example();
11
12 defparam U0.my_secret = 11;
13 defparam U1.my_secret = 22;
14
15 secret_number U0();
16 secret_number U1();
17
18 endmodule
```

You could download file defparam\_example.v [here](#)



### Parameter Override during instantiating.

```
1 module secret_number;
2 parameter my_secret = 0;
3
4 initial begin
5     $display("My secret number in module is %d", my_secret);
6 end
7
8 endmodule
9
```

```

10 module param_override_instance_example();
11
12 secret_number #(11) U0();
13 secret_number #(22) U1();
14
15 endmodule

```

You could download file param\_override\_instance\_example.v [here](#)



### Passing more than one parameter

```

1 module ram_sp_sr_sw (
2   clk           , // Clock Input
3   address       , // Address Input
4   data          , // Data bi-directional
5   cs            , // Chip Select
6   we            , // Write Enable/Read Enable
7   oe            , // Output Enable
8 );
9
10 parameter DATA_WIDTH = 8 ;
11 parameter ADDR_WIDTH  = 8 ;
12 parameter RAM_DEPTH = 1 << ADDR_WIDTH;
13 // Actual code of RAM here
14
15 endmodule

```

You could download file param\_more\_then\_one.v [here](#)

When instantiating more than the one parameter, parameter values should be passed in the order they are declared in the sub module.

```

1 module ram_controller (); //Some ports
2
3 // Controller Code
4
5 ram_sp_sr_sw #(16,8,256) ram(clk,address,data,cs,we,oe);
6
7 endmodule

```

You could download file param\_more\_then\_one1.v [here](#)



### Verilog 2001

In Verilog 2001, the code above will work, but the new feature makes the code more readable and error free.

```

1 module ram_controller (); //Some ports

```



```

2
3 ram_sp_sr_sw #(
4     .DATA_WIDTH(16) ,
5     .ADDR_WIDTH(8) ,
6     .RAM_DEPTH(256))  ram(clk,address,data,cs,we,oe) ;
7
8  endmodule
  
```

You could download file param\_more\_than\_one2.v [here](#)

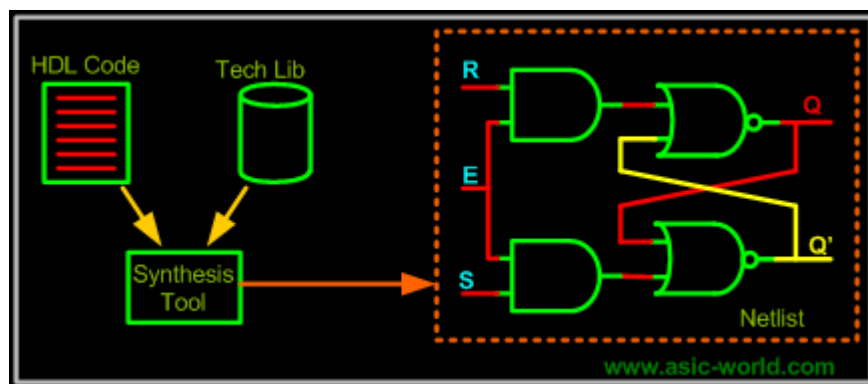
Was this copied from VHDL?

### What is logic synthesis ?

Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation. Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, muxes, memory, and flip-flops. Standard cells put together are called technology library. Normally the technology library is known by the transistor size (0.18u, 90nm).

A circuit description is written in Hardware Description Language (HDL) such as Verilog. The designer should first understand the architectural description. Then he should consider design constraints such as timing, area, testability, and power.

We will see a typical design flow with a large example in the last chapter of Verilog tutorial.





### Life before HDL (Logic synthesis)

As you must have experienced in college, everything (all the digital circuits) is designed manually. Draw K-maps, optimize the logic, draw the schematic. This is how engineers used to design digital logic circuits in early days. Well this works fine as long as the design is a few hundred gates.



### Impact of HDL and Logic synthesis.

High-level design is less prone to human error because designs are described at a higher level of abstraction. High-level design is done without significant concern about design constraints. Conversion from high-level design to gates is done by synthesis tools, using various algorithms to optimize the design as a whole. This removes the problem with varied designer styles for the different blocks in the design and suboptimal designs. Logic synthesis tools allow technology independent design. Design reuse is possible for technology-independent descriptions.



### What do we discuss here ?

When it comes to Verilog, the synthesis flow is the same as for the rest of the languages. What we try to look in next few pages is how particular code gets translated to gates. As you must have wondered while reading earlier chapters, how could this be represented in Hardware ? An example would be "delays". There is no way we could synthesize delays, but of course we can add delay to particular signals by adding buffers. But then this becomes too dependent on synthesis target technology. (More on this in the VLSI section).

First we will look at the constructs that are not supported by synthesis tools; the table below shows the constructs that are not supported by the synthesis tool.



### Constructs Not Supported in Synthesis

| Construct Type | Notes   |
|----------------|---|
| initial        | Used only in test benches.                                |
| events         | Events make more sense for syncing test bench components. |
| real           | Real data type not supported.                             |
| time           | Time data type not supported.                             |

|                     |  |
|---------------------|--|
| force and release   | Force and release of data types not supported.   |
| assign and deassign | assign and deassign of reg data types is not supported. But assign on wire data type is supported. |
| fork join           | Use nonblocking assignments to get same effect.  |
| primitives          | Only gate level primitives are supported.  |
| table               | UDP and tables are not supported.  |



### Example of Non-Synthesizable Verilog construct.

Any code that contains the above constructs are not synthesizable, but within synthesizable constructs, bad coding could cause synthesis issues. I have seen codes where engineers code a flip-flop with both posedge of clock and negedge of clock in sensitivity list.

Then we have another common type of code, where one reg variable is driven from more than one always block. Well it will surely cause synthesis error.



### Example - Initial Statement

```

1 module synthesis_initial(
2   clk,q,d);
3 input  clk,d;
4 output q;
5 reg    q;
6
7 initial begin
8   q <= 0;
9 end
10
11 always @ (posedge clk)
12 begin
13   q <= d;
14 end
15
16 endmodule
  
```

You could download file synthesis\_initial.v [here](#)



### Delays

a = #10 b; This code is useful only for simulation purpose.

Synthesis tool normally ignores such constructs, and just assumes that there is no #10 in above statement, thus treating above code as

a = b;



### Comparison to X and Z are always ignored

```

1 module synthesis_compare_xz (a,b);
2 output a;
3 input b;
4 reg a;
5
6 always @ (b)
7 begin
8   if ((b == 1'bz) || (b == 1'bx)) begin
9     a = 1;
10  end else begin
11    a = 0;
12  end
13 end
14
15 endmodule
  
```

You could download file synthesis\_compare\_xz.v [here](#)

There seems to be a common problem with all the design engineers new to hardware. They normally tend to compare variables with X and Z. In practice it is the worst thing to do, so please avoid comparing with X and Z. Limit your design to two states, 0 and 1. Use tri-state only at chip IO pads level. We will see this as an example in the next few pages.



### Constructs Supported in Synthesis

Verilog is such a simple language; you could easily write code which is easy to understand and easy to map to gates. Code which uses if, case statements is simple and cause little headaches with synthesis tools. But if you like fancy coding and like to have some trouble, ok don't be scared, you could use them after you get some experience with Verilog. Its great fun to use high level constructs, saves time.

The most common way to model any logic is to use either assign statements or always blocks. An assign statement can be used for modeling only combinational logic and always can be used for modeling both combinational and sequential logic.

| Construct Type | Keyword or Description | Notes |
|----------------|------------------------|-------|
|----------------|------------------------|-------|

|                       |   |   |
|-----------------------|---|---|
| ports                 | input, inout, output                        | Use inout only at IO level.   |
| parameters            | parameter                                   | This makes design more generic  |
| module definition     | module                                      |   |
| signals and variables | wire, reg, tri                              | Vectors are allowed   |
| instantiation         | module instances / primitive gate instances | E.g.- nand (out,a,b), bad idea to code RTL this way.                  |
| function and tasks    | function , task                             | Timing constructs ignored   |
| procedural            | always, if, else, case, casex, casez        | initial is not supported  |
| procedural blocks     | begin, end, named blocks, disable           | Disabling of named blocks allowed                                     |
| data flow             | assign                                      | Delay information is ignored  |
| named Blocks          | disable                                     | Disabling of named block supported.                                   |
| loops                 | for, while, forever                         | While and forever loops must contain @(posedge clk) or @(negedge clk) |



### Operators and their Effect.

One common problem that seems to occur is getting confused with logical and reduction operators. So watch out.

| Operator Type | Operator Symbol | Operation Performed |
|---------------|-----------------|---------------------|
| Arithmetic    | *               | Multiply            |
|               | /               | Division            |
|               | +               | Add                 |
|               | -               | Subtract            |
|               | %               | Modulus             |
|               | +               | Unary plus          |
|               | -               | Unary minus         |
| Logical       | !               | Logical negation    |
|               | &&              | Logical AND         |

|                      |                       |                       |
|----------------------|-----------------------|-----------------------|
|                      | <code>  </code>       | Logical OR            |
| <b>Relational</b>    | <code>&gt;</code>     | Greater than          |
|                      | <code>&lt;</code>     | Less than             |
|                      | <code>&gt;=</code>    | Greater than or equal |
|                      | <code>&lt;=</code>    | Less than or equal    |
| <b>Equality</b>      | <code>==</code>       | Equality              |
|                      | <code>!=</code>       | inequality            |
| <b>Reduction</b>     | <code>&amp;</code>    | Bitwise AND           |
|                      | <code>~&amp;</code>   | Bitwise NAND          |
|                      | <code> </code>        | Bitwise OR            |
|                      | <code>~ </code>       | Bitwise NOR           |
|                      | <code>^</code>        | Bitwise XOR           |
|                      | <code>^~ ~^</code>    | Bitwise XNOR          |
| <b>Shift</b>         | <code>&gt;&gt;</code> | Right shift           |
|                      | <code>&lt;&lt;</code> | Left shift            |
| <b>Concatenation</b> | <code>{ }</code>      | Concatenation         |
| <b>Conditional</b>   | <code>?</code>        | conditional           |

### Logic Circuit Modeling

From what we have learnt in digital design, we know that there could be only two types of digital circuits. One is combinational circuits and the second is sequential circuits. There are very few rules that need to be followed to get good synthesis output and avoid surprises.



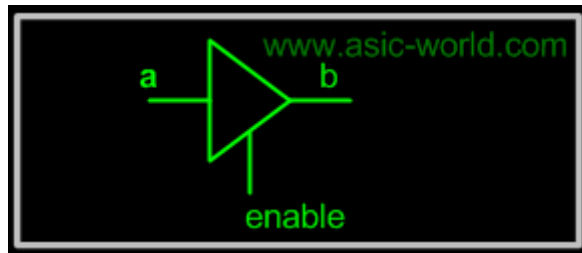
#### Combinational Circuit Modeling using assign

Combinational circuits modeling in Verilog can be done using assign and always blocks. Writing simple combinational circuits in Verilog using assign statements is very straightforward, like in the example below

```
assign y = (a&b) | (c^d);
```



#### Tri-state buffer

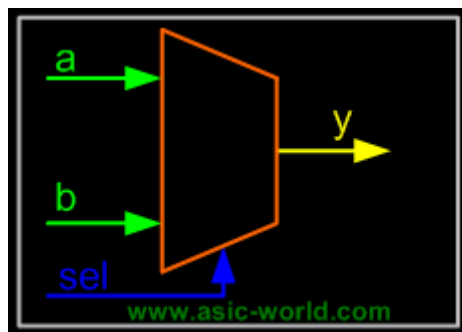


```

1 module tri_buf (a,b,enable);
2   input a;
3   output b;
4   input enable;
5   wire a,enable;
6   wire b;
7
8   assign b = (enable) ? a : 1'bz;
9
10  endmodule
  
```

You could download file tri\_buf.v [here](#)

## ✦ Mux

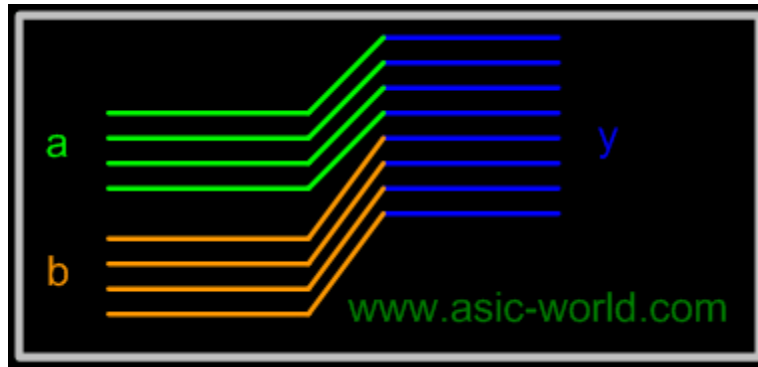


```

1 module mux_21 (a,b,sel,y);
2   input a, b;
3   output y;
4   input sel;
5   wire y;
6
7   assign y = (sel) ? b : a;
8
9   endmodule
  
```

You could download file mux\_21.v [here](#)

## ✦ Simple Concatenation



```

1 module bus_con (a,b);
2     input  [3:0] a, b;
3     output [7:0] y;
4     wire  [7:0] y;
5
6     assign y = {a,b};
7
8 endmodule

```

You could download file bus\_con.v [here](#)

### ✦ 1 bit adder with carry

```

1 module addbit (
2     a      , // first input
3     b      , // Second input
4     ci     , // Carry input
5     sum    , // sum output
6     co     , // carry output
7 );
8 //Input declaration
9 input a;
10 input b;
11 input ci;
12 //Ouput declaration
13 output sum;
14 output co;
15 //Port Data types
16 wire a;
17 wire b;
18 wire ci;
19 wire sum;
20 wire co;
21 //Code starts here

```



```

22 assign {co,sum} = a + b + ci;
23
24 endmodule // End of Module addbit

```

You could download file addbit.v [here](#)

### ✦ Multiply by 2

```

1 module multiply (a,product);
2     input [3:0] a;
3     output [4:0] product;
4     wire [4:0] product;
5
6     assign product = a << 1;
7
8 endmodule

```

You could download file multiply.v [here](#)

### ✦ 3 is to 8 decoder

```

1 module decoder (in,out);
2 input [2:0] in;
3 output [7:0] out;
4 wire [7:0] out;
5 assign out = (in == 3'b000 ) ? 8'b0000_0001 :
6 (in == 3'b001 ) ? 8'b0000_0010 :
7 (in == 3'b010 ) ? 8'b0000_0100 :
8 (in == 3'b011 ) ? 8'b0000_1000 :
9 (in == 3'b100 ) ? 8'b0001_0000 :
10 (in == 3'b101 ) ? 8'b0010_0000 :
11 (in == 3'b110 ) ? 8'b0100_0000 :
12 (in == 3'b111 ) ? 8'b1000_0000 : 8'h00;
13
14 endmodule

```

You could download file decoder.v [here](#)



### Combinational Circuit Modeling using always

While modeling using always statements, there is the chance of getting a latch after synthesis if care is not taken. (No one seems to like latches in design, though they are faster, and take lesser transistor. This is due to the fact that timing analysis tools always have problems with latches; glitch at enable pin of latch is another problem).

One simple way to eliminate the latch with always statement is to always drive 0 to the LHS variable in the beginning of always code as shown in the code below.

### ✦ 3 is to 8 decoder using always

```
1 module decoder_always (in,out);
2 input [2:0] in;
3 output [7:0] out;
4 reg [7:0] out;
5
6 always @ (in)
7 begin
8     out = 0;
9     case (in)
10         3'b001 : out = 8'b0000_0001;
11         3'b010 : out = 8'b0000_0010;
12         3'b011 : out = 8'b0000_0100;
13         3'b100 : out = 8'b0000_1000;
14         3'b101 : out = 8'b0001_0000;
15         3'b110 : out = 8'b0100_0000;
16         3'b111 : out = 8'b1000_0000;
17     endcase
18 end
19
20 endmodule
```

You could download file decoder\_always.v [here](#)



## Sequential Circuit Modeling

Sequential logic circuits are modeled using edge sensitive elements in the sensitive list of always blocks. Sequential logic can be modeled only using always blocks. Normally we use nonblocking assignments for sequential circuits.

### ✦ Simple Flip-Flop

```
1 module flif_flop (clk,reset, q, d);
2 input clk, reset, d;
3 output q;
4 reg q;
5
6 always @ (posedge clk )
7 begin
8     if (reset == 1) begin
```

```
9      q <= 0;  
10     end else begin  
11       q <= d;  
12     end  
13 end  
14  
15 endmodule
```

You could download file flip\_flop.v [here](#)



### Verilog Coding Style

If you look at the code above, you will see that I have imposed a coding style that looks cool. Every company has got its own coding guidelines and tools like linters to check for this coding guidelines. Below is a small list of guidelines.

- Use meaningful names for signals and variables
- Don't mix level and edge sensitive elements in the same always block
- Avoid mixing positive and negative edge-triggered flip-flops
- Use parentheses to optimize logic structure
- Use continuous assign statements for simple combo logic
- Use nonblocking for sequential and blocking for combo logic
- Don't mix blocking and nonblocking assignments in the same always block (even if Design compiler supports them!!).
- Be careful with multiple assignments to the same variable
- Define if-else or case statements explicitly

**Note :** Suggest if you want more details.